Università degli Studi di Bologna
Scuola di Ingegneria e Architettura

APPUNTI DI SISTEMI OPERATIVI T

Corso di Laurea in Ingegneria Informatica – A.A. 2014-2015

```
#!/bin/bash
if [ -f prova.txt ]; then
    echo "Il file prova.txt esiste."
else
    echo "Il file prova.txt non esiste o non è un file"
fi
```

Docente **ANNA CIAMPOLINI**
Appunti di **ALESSANDRO CELSO**
Corso di Sistemi Operativi T

Abstract
Tale documento altro non è che una revisione e riveduta generale di un blocco appunti condiviso da uno studente del corso di ing. Informatica presso l'Università di Bologna, blocco appunti purtroppo anonimo (avrei gradito dargli i crediti per l'eccellente lavoro svolto), a cui ho aggiunto diverse note, in particolare un intero capitolo dedicato allo scripting bash, da me prese durante il corso di Sistemi Operativi T, tenuto dal docente Ciampolini Anna nel corso dell'anno accademico 2014-2015.
Tale documento non è assolutamente considerabile un blocco di dispense ufficiale, tantomeno sostitutivo delle slides, a cui consiglio sempre di attenersi per lo studio.
Per suggerimenti o eventuali modifiche scrivere a **cttynul[at]redchan[dot]it** *o* **cttynul[at]gmail[dot]com**

Ultimo aggiornamento: *Ottobre 2015*

1.0 SISTEMA OPERATIVO

E' un programma, insieme di programmi, che agisce da intermediario tra l'utente e l'hardware del PC

- Fornisce all'utente una visione astratta e semplificata dell'hardware.
- Gestisce in modo efficace ed efficiente le risorse del sistema.

Il sistema operativo interfaccia programmi applicativi o di sistema con le risorse hardware (CPU, memoria centrale, memoria secondaria, connessioni di rete). Inoltre, mappa le risorse hardware in risorse logiche, accessibili attraverso interfacce ben definite (processi, file system, memoria virtuale).

1.0.1 EVOLUZIONE DEI SISTEMI OPERATIVI

1. **Prima generazione**: controllo del sistema completamente manuale, linguaggio macchina. Non è presente alcun sistema operativo

2. **Seconda generazione**: sistemi batch semplici, linguaggio di alto livello, possibilità di effettuare operazioni di input mediante delle schede perforate. Aggregazioni di programmi in lotti (batch = insieme di programmi(job) eseguiti in modo sequenziale). In questo tipo di sistemi il compito unico del sistema operativo era quello di trasferire il controllo da un job, appena terminato, al prossimo da eseguire. Il sistema operativo risiede in memoria (monitor). I principali svantaggi sono: l'assenza di interazione con l'utente, inattività della CPU causata dalla sospensione di un job che attende un evento, sequenza.

3. **Multiprogrammazione**: per far fronte ai problemi sopra descritti, nascono i sistemi batch multiprogrammati: viene precaricato sul disco un insieme di job (pool). Il sistema operativo ha il compito di caricare in memoria centrale un sottoinsieme dei job precaricati. Tra i job che il sistema operativo ha caricato in memoria centrale, ne viene selezionato uno a cui verrà assegnata la CPU. Qualora il job corrente si pone in attesa di un evento, il sistema operativo assegna la CPU ad un altro job

4. **Sistemi time-sharing**: Nascono della necessità di avere dei sistemi che permettano maggiore interattività con l'utente, e la possibilità di gestire più utenti che interagiscono contemporaneamente con il sistema operativo (multi-utenza).

    - *Multi-utenza*: il sistema presenta ad ogni utente una macchina virtuale completamente dedicata, in termini di: utilizzo della CPU e di utilizzo di altre risorse.
    - *Interattività*: per garantire un accettabile velocità di "reazione" alle richieste dei singoli utenti, il sistema operativo interrompe l'esecuzione di un job dopo un intervallo di tempo prefissato (timeslice), assegnando la CPU ad un altro job.

1.0.2 FUNZIONAMENTO A INTERRUZIONI

Le varie componenti hardware e software del sistema interagiscono con il sistema operativo attraverso delle interruzioni asincrone (Interrupt). Per la precisione ogni interruzione è causata da un evento (richiesta di servizi al SO, completamento operazioni I/O, accesso non consentito alla memoria). Ad ogni interruzione è associata una routine di servizio (handler)

per la gestione dell'evento.

1. **Interruzioni hardware**: i dispositivi inviano segnali alla CPU per notificare particolari eventi al sistema operativo.

2. **Interruzioni software**: i programmi in esecuzione possono generare delle interruzioni: quando tentano l'esecuzione di operazioni non lecite oppure quando richiedono l'esecuzione di servizi al sistema operativo (System call)

Non appena il sistema operativo riceve un interruzione, interrompe la sua esecuzione, salvando lo stato in memoria, attiva la routine di servizio (handler) ed infine ripristina lo stato precedentemente salvato.

NB: Per individuare le routine di servizio adeguata, si utilizza un *vettore delle interruzioni*.

1.2.1 PROTEZIONE

Nei sistemi che prevedono multiprogrammazione e multiutenza sono necessari alcuni meccanismi di protezione. Nello specifico, le risorse allocate a programmi o utenti devono essere protette nei confronti di accessi illeciti di altri programmi/utenti.

Per prima cosa è necessario impedire al programma in esecuzione di accedere ad aree di memoria esterne al proprio spazio. Ogni job infatti, ha un suo proprio spazio di indirizzi.

Per garantire protezione, molte architetture di CPU prevedono un duplice modo di funzionamento:

1. User mode: tutti i programmi utente sono gestiti in user mode.

2. Kernel mode: può eseguire istruzioni privilegiate (sono le più "pericolose": accesso ai dispositivi I/O, gestione della memoria, disabilitazione interruzioni...) Il sistema operativo esegue in modo kernel, a differenza dei programmi utente che per l'esecuzione di un istruzione privilegiata devono ricorrere ad una *system call.*

    - *System call:* Il programma invia un interruzione software al sistema operativo, dopo aver salvato il suo stato, il sistema esegue in modo kernel l'operazione richiesta. Una volta terminata l'operazione, il controllo ritorna al programma chiamante (user mode)

1.3 STRUTTURA DEL SISTEMA OPERATIVO

Le componenti principali di un sistema operativo:

- **Gestione dei processi.**
  1. Creazione/terminazione dei processi
  2. Sospensione/ripristino dei processi
  3. Sincronizzazione/comunicazione dei processi.

- **Gestione della memoria centrale CPU**
  1. Separare gli spazi di indirizzi associati ai processi
  2. Allocare/deallocare memoria dei processi.
  3. *Memoria virtuale:* gestire spazi logici di indirizzi di dimensioni complessivamente superiori allo spazio fisco
  4. Realizzazione di collegamenti tra memoria logica e memoria fisica.

- **Gestione della memoria secondaria e del file system**
  1. Allocazione/deallocazione di spazio
  2. Gestione dello spazio libero
  3. Scheduling delle operazioni su disco
  4. Fornire una visione logica uniforme della memoria secondaria:
     - Realizzare il concetto astratto di file, come unità di memorizzazione logica
     - Fornire una struttura astratta per l'organizzazione dei file
  5. Creazione/cancellazione file
  6. Manipolazione dei file
  7. Associazione tra file e dispositivi di memorizzazione secondaria

- **Gestione I/O.**
  1. Interfaccia tra programmi e dispositivi
  2. Per ogni dispositivo: *device driver. (*Routine per l'interazione con un particolare dispositivo)

- **Protezione e sicurezza**
  1. Controllo dell'accesso alle risorse da parte dei processi mediante: autorizzazioni e modalità di accesso.

- **Interfaccia utente**
  1. Interprete dei comandi *shell* linea di comando.
  2. Interfaccia grafica (GUI) = interazione mouse - elementi grafici. Di solito è organizzata a finestre.

Le componenti sopra elencate possono essere organizzate in modi differenti all'interno del sistema operativo:

1. **Struttura monolitica**: il sistema operativo è costituito da un unico modulo contenente un insieme di procedure, che realizzano le varie componenti. Il principale vantaggio di un sistema monolitico è il basso costo di interazione tra le varie componenti, ma il SO è un sistema complesso e presenta gli stessi requisiti delle applicazioni in the large.

2. **Struttura modulare**: le varie componenti del sistema operativo vengono organizzate in moduli caratterizzati da interfacce ben definite. Ogni strato ha un insieme di funzionalità che vengono offerte allo strato superiore mediante interfacce.

    - VANTAGGI:
        - *Astrazione:* ogni livello è un oggetto astratto, che fornisce ai livelli superiori una visione astratta del sistema.
        - *Modularità:* possibilità di sviluppo, verifica, modifica in modo indipendente dagli altri livelli.
    - SVANTAGGI:
        - *Organizzazione gerarchica* tra le componenti, non sempre è possibile, difficoltà di realizzazione.
        - *Scarsa efficienza:* costo di attraversamento dei livelli.

3. **Microkernel**: La struttura del nucleo (kernel) è ridotta a poche funzionalità di base mentre il resto del sistema operativo è rappresentato da processi utente. E' un sistema affidabile e personalizzabile ma non molto efficiente a causa delle molte chiamate a system call.

2.0 SCHEDULING

E' l'attività mediante la quale il sistema operativo effettua delle scelte tra i processi, riguardo a:

1. Caricamento in memoria centrale.
2. Assegnazione alla CPU.

In generale, il sistema operativo compie tre diverse attività di scheduling:

- **Scheduling a lungo termine**:
  E' quella componente del sistema operativo che si occupa di selezionare i programmi presenti nella memoria secondaria da caricare nella memoria centrale (creando i corrispondenti processi)

NB: Nei sistemi timesharing non è presente; molto spesso, infatti, è l'utente che stabilisce il grado di multiprogrammazione.

- **Scheduling a medio termine (Swapping):**
  Si occcupa di trasferire in memoria secondaria, temporaneamente, dei processi, o parti di essi, in modo da consentire il caricamento di altri processi.

- **Scheduling a breve termine (CPU):**
  E' quella parte del sistema operativo che si occupa della selezione dei processi a cui assegnare la CPU. Una volta selezionato il processo, una parte del sistema operativo (**Dispatcher**) effettuerà un cambio di contesto (**context switch)**

Gestisce le code dei processi pronti: Contiene i PCB dei processi in stato di ready.
Cambio di contesto: E' la fase in cui l'uso della CPU viene commutato da un processo ad un altro.
Quando si verifica questo cambio di contesto, il sistema operativo deve salvare lo stato del processo uscente, aggiornando il suo PCB e trasferire i dati dal PCB del processo entrante alla CPU.

2.1 OPERAZIONE SUI PROCESSI

Ogni sistema operativo multiprogrammato prevede dei meccanismi per la gestione dei processi:

1. Creazione
2. Interazione tra processi
3. Terminazione

Queste, sono delle operazioni privilegiate (esecuzione in modo kernel)

2.2 PROCESSI LEGGERI (THREAD)

Un thread è un unità di esecuzione che condivide codice e dati con altri thread ad esso associati.

TASK: insieme di thread che riferiscono lo stesso codice e gli stessi dati (il codice e i dati non sono caratteristiche del singolo thread, ma del task al quale appartengono)

A differenza dei processi pesanti, un thread può condividere variabili con altri, questo porta ad un minor costo di context switch, poiché i PCB dei thread non contengono alcuna informazione relativa a codice e dati. Un piccolo svantaggio dell'utilizzo dei thread è a sicurezza, in quanto un thread può modificare i dati di un altro che appartiene alla stessa task.

2.2.1 PROCESSI INDIPENDENTI ED INTERAGENTI

- *Processi interagenti*: due processi sono interagenti se l'esecuzione di uno è influenzata dall'esecuzione di un altro e/o viceversa. Esistono diversi tipi di interazione:
    - **Cooperazione**: Interazione prevedibile e desiderata, i processi collaborano per il raggiungimento di un fine comune.
    - **Competizione**: Interazione prevedibile ma "non desiderata".
    - **Interferenza**: Interazione non prevista e non desiderata.

  L'interazione può avvenire mediante:

  1. **Memoria condivisa** (Ambiente globale): Il sistema operativo consente ai processi di condividere variabili: l'interazione avviene tramite l'accesso a variabili condivise.
  2. **Scambio di messaggi** (Ambiente locale): I processi non condividono variabili e interagiscono mediante trasmissione/ricezione di messaggi.

Il vantaggio principale di utilizzare processi che posso interagire fra di loro (condivisione di informazioni) è la velocità di esecuzione, in quanto vi è una suddivisione dei compiti tra i vari processi (Modularità).

- *Processi indipendenti:* due processi si dicono indipendenti se l'esecuzione di uno non è influenzata dall'altro e viceversa.

3.0 UNIX

E' un sistema operativo multiprogrammatato a divisione di tempo. Unità di computazione è il **processo**

Caratteristiche del processo UNIX*:* Ogni processo ha un proprio spazio di indirizzamento completamente locale e non condiviso. Modello ad ambiente locale (o a scambio di messaggi).

NB: **il codice** può essere condiviso (codice rientrante)

Stati di un processo UNIX*:* Oltre agli stati generali (init, ready, running, sleeping, terminated) si hanno in aggiunta due nuovi:

- **Zombie**: il processo è terminato, ma è in attesa che il padre ne riveli lo stato di terminazione
- **Swapped**: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria dallo scheduler a medio termine che effettua uno *swap out* (si applica preferibilmente ai processi più lunghi o a quelli bloccati)

Rappresentazione processi UNIX*:* Il codice dei processi è rientrante, quindi più processi possono condividere lo stesso codice (text). Per permettere questo il codice ed i dati devono essere separati (modello a codice puro). Il sistema operativo gestisce una struttura dati globale in cui sono contenuti i puntatori ai codici utilizzati, eventualmente condivisi, dai processi. Questa struttura viene chiamata: **text table.** L'elemento della text table si chiama **text structure** e contiene: un puntatore al codice ed il numero dei processi che lo condividono.

A differenza di quanto abbiamo visto prima, la PCB è rappresentata da due strutture dati (invece di una):

1. **Process structure:** contiene le informazioni necessarie al sistema per la gestione del processo (id, stato, puntatori alle varie aree dati, riferimento all'elemento della text table associato al codice del processo, informazioni di scheduling...). Le process structure sono organizzate in un vettore: **process table.**
2. **User structure:** informazioni necessarie solo se il processo è residente in memoria centrale (registri CPU, informazioni sulle risorse allocate, informazioni sulla gestione dei segnali, ambiente del processo...)

Immagine di un processo UNIX: E' l'insieme di aree di memoria e strutture dati, associate al processo. ATTENZIONE: non tutta l'immagine è accessibile in modo user. Ogni processo può essere soggetto a swapping: ma non tutta l'immagine di un processo può essere trasferita in memoria: una parte swappable ed una parte residente non swappable.

3.0.1 SYSTEM CALL

- ***Creazione di processi:***

```
int fork(void);
```

Questa funzione consente a un processo di generare un processo figlio.

1. Padre e figlio condividono lo stesso codice. Il figlio eredita una copia dei dati del padre.

2. La fork non richiede parametri e restituisce un intero:
    1. 0 per il processo figlio creato.
    2. >0 (PID) per il processo padre
    3. <0 se la creazione non è andata a buon fine.

3. L'utilizzo della fork() comporta l'allocazione di una nuova process structure nella process table associata al processo figlio e l'allocazione di una nuova user structure nella quale viene copiata la user structure del padre. Inoltre devono essere allocati i segmenti di dati e stack del figlio, nei quali verranno copiati quelli del padre.

4. Dopo aver utilizzato una fork():
    1. Padre e figlio procedono in parallelo
    2. Ogni variabile del figlio è inizializzata con il valore assegnatole dal padre prima della fork()
    3. Le risorse allocate al padre sono condivise con i figli

- ***Terminazione dei processi:***

```
void exit(int status);
```

Questa funzione prevede un parametro **status** mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione

1. L'utilizzo della exit() comporta la chiusura dei file aperti non condivisi e la terminazione del processo:
    - Se il processo che termina ha figli in esecuzione, il processo init adotta i figli dopo la terminazione del padre.
    - Se il processo termina prima che il padre ne riveli lo stato di terminazione con la system call wait(), il processo passa nello stato zombie.

- ***Sospensione in attesa della terminazione dei figli:***

```
int wait(int * status);
```

Questa funzione prevede un parametro "status" che rappresenta l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio

- Il risultato prodotto dalla wait() è il PID del processo terminato oppure un codice di errore (<0)

    1. Il processo che chiama la wait() può avere figli in esecuzione:
        1. Se tutti i figli non sono ancora terminati, il processo si sospende in attesa della terminazione del primo di esso.
        2. Se almeno un figlio è già terminato ed il suo stato non è stato ancora rilevato (zombie) la wait() ritorna immediatamente con il suo stato di terminazione
        3. Se non esiste neanche un figlio (LA WAIT() NON E' SOSPENSIVA) ritorna un codice di errore

NB: In caso di terminazione di un figlio, la variabile status raccoglie lo stato di terminazione. Nel caso in cui il byte meno significativo rappresenti lo stato di terminazione si parla di terminazione volontaria (exit), in caso contrario il byte meno significativo descrive il segnale che ha terminato il figlio (terminazione involontaria)

Lo standard POSIX.1 prevede delle macro (definite nell'header file <sys/wait.h>) per l'analisi dello stato di terminazione. In particolare:

- **WIFEXITED(status):** restituisce vero se il processo figlio è terminato volontariamente (EXIT_SUCCESS). In questo caso la macro
    - ***WEXITSTATUS(status)*** restituisce lo stato di terminazione
- **WIFSIGNALED(status**): restituisce vero se il processo figlio è terminato involontariamente. In questo caso la macro
    - ***WTERMSIG(status)*** restituisce il numero del segnale che ha causato la terminazione

Per convenzione in C viene indicato VERO = 1 e FALSO = 0.

- ***Sostituzione di codice e dati***:

  ◦ `exec()`

  ◦ Ci permette di differenziare il codice di due processi. In altre parole, vengono sostituiti il codice e gli argomenti di invocazione del processo chiamante, con il codice e gli argomenti di un programma specificato come parametro della funzione.
  NB NON GENERA NUOVI PROCESSI

  1. L'utilizzo di questa funzione comporta:
     - Codice, dati globali, stack e heap nuovi
     - Riferisce un nuovo text
     - Mantiene la stessa process structure

  2. In UNIX è possibile differenziare il codice di due processi mediante una system call della famiglia exec()
     - `execl(), execle(), execlp, execv(), execve(), execvp()…`

```
int execl (char * pathname, char *arg0, … , char * argN, (char*)0);
```

- **pathname**: è il nome dell'eseguibile da caricare.
- **Arg0** è il nome del programma
- **arg1,...,argN** sono gli argomenti da passare al programma
- **(char*)0** è il puntatore nullo che termina la lista

Es. `excel("/bin/ls","ls", "-l", "pippo", (char*)0);`

```
int execve(char *pathname, char *argV[], , char * env[]);
```

- **pathname**: è il nome dell'eseguibile da caricare.
- **argv** è il vettore degli argomenti del programma da eseguire
- **env** è il vettore delle variabili di ambiente da sostituire all'ambiente del processo (contiene stringhe del tipo "VARIABILE=valore")

- ***Gestione degli errori***

```
void perror("stringa")
```

- In caso di fallimento ogni system call restituisce un valore negativo tipicamente -1
- In aggiunta, UNIX prevede la variabile globale **errno**, alla quale il kernel assegna il codice di errore generato dall'ultima systemcall eseguita. Per

interpretare il valore è possibile usare la system call perror();

4.0 SCHEDULING DELLA CPU

L'obbiettivo principale della multiprogrammazione è la massimizzazione dell'utilizzo della memoria centrale: lo scheduling della CPU si occupa proprio di questo; commutare l'utilizzo della memoria tra i vari processi.

Scheduler della CPU: E' quella parte del sistema operativo che selezione dalla coda dei processi in stato di ready, il prossimo processo al quale assegnare l'uso della CPU.
La gestione della coda dei processi è realizzata mediante *politiche di scheduling.*

Gli algoritmi di scheduling si possono classificare in due categorie:

- **Senza prelazione (non pre-emptive)**: La CPU rimane allocata al processo running finché esso non si sospende volontariamente o non termina

- **Con prelazione (pre-emptive)**: processo running può essere **prelazionato**, cioè SO può sottrargli CPU per assegnarla ad un nuovo processo

NB: I sistemi a divisione di tempo hanno sempre uno scheduling **pre-emptive**

4.0.1 CRITERI DI SCHEDULING: per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni *indicatori di performance:*

5. **Utilizzo della CPU:** percentuale media di utilizzo CPU nell'unità di tempo
6. **Throughput** (del sistema): numero di processi completati nell'unità di tempo
7. **Tempo di Attesa** (di un processo): tempo totale trascorso nella ready queue
8. **Turnaround** (di un processo): tempo tra la sottomissione del job e il suo completamento
9. **Tempo di Risposta** (di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta

In generale, devono essere massimizzati "utilizzo della CPU" e "throughpu" e minimizzati i restani.
NB: non è possibile ottimizzare tutti i criteri contemporaneamente, a seconda del sistema operativo, le politiche di scheduling possono avere diversi obiettivi

4.0.2 POLITICHE DI SCHEDULING

- **FCFS_first come first served:** la coda dei processi pronti è gestita in modalità FIFO. E' un algoritmo non pre-emptive, non è possibile influire sull'ordine dei processi.

- **SJF_shortest job first:** per ogni processo nella ready queue (coda dei processi pronti) viene stimata la lunghezza del prossimo CPU-burst, viene schedulato il processo con il CPU burst più corto. Questo algoritmo ottimizza il tempo di attesa e può essere: non pre-emptive e **pre-emptive** (se nella coda arriva un processo con CPU burst minore del CPU burst rimasto al processo in stato di running)

- **Round Robin:** Tipicamente utilizzata nei sistemi time sharing: la coda dei processi è

gestita come una coda **FIFO circolare.** Ad ogni processo viene allocata la CPU per un intervallo di tempo costante (time slice), scaduto questo tempo il processo viene re-inserito in coda e la CPU passa ad un altro processo. La RR può essere vista come un estensione del FCFS.

- **Con priorità:** Ad ogni processo vine assegnata una priorità, che può essere:
  - **Definita internamente:** il sistema operativo attribuisce ad ogni processo una priorità in base a politiche interne
  - **Esternamente:** criteri esterni al sistema operativo.

Le priorità possono essere costanti o variare dinamicamente. Questo tipo di algoritmo presenta un problema: **starvation dei processi.** In altre parole, la starvation si verifica quando uno o più processi di priorità bassa vengono lasciati indefinitamente nella coda dei processi pronti, perché vi è sempre un processo pronto di priorità più alta. La soluzione a questo problema è la **modifica dinamica delle priorità**, come ad esempio, la priorità decresce al crescere del tempo di CPU giù utilizzato, mente cresce dinamicamente con il tempo di attesa del processo.

NB: Nei sistemi operativi reali, **spesso si combinano diversi algoritmi di scheduling.**

APPROFONDIMENTI:

- **Scheduling di UNIX**: privilegiare i processi interattivi, aggiornamento dinamico delle priorità. (L'utente può influire sulla priorità: comando nice; ovviamente soltanto per decrescere la priorità)
- **Scheduling dei thread Java:** JVM (java virtual machine) usa scheduling con prelazione e basato su priorità. Non specifica se i thread hanno un quantitativo di tempo oppure no. Siccome non garantisce time-slicing , andrebbe usato il metodo yield(), per trasferire il controllo ad un altro thread di uguale priorità.

4.0.3 COMUNICAZIONE TRA PROCCESSI

Il sistema operativo offre dei meccanismi a supporto della comunicazione tra i processi, quali, **send** (predizione di messaggi) e **receive** (ricezione di messaggi).

Comunicazione diretta: al messaggio viene associato l'identificatore del processo destinatario. I due processi devono conoscersi reciprocamente così da creare automaticamente il canale di comunicazione. Questo tipo di approccio presenta una scarsa modularità, in quanto, la modifica di un processo implica la revisione di tutte le operazioni di comunicazione (difficoltà di riutilizzo)

Comunicazione indiretta: il messaggio viene indirizzato ad una mailbox, dalla quale il destinatario potrà prelevarlo. I processi non sono tenuti a conoscersi in quanto i messaggi verranno depositati/prelevati direttamente dalla mailbox (canale di comunicazione).

> *Mailbox:* è una risorsa astratta condivisibile da più processi. Funge da contenitore messaggi.

Il vantaggio di utilizzare questo metodo è il poter associare il canale a più di due processi.

Canale di comunicazione: lo scambio di messaggi avviene mediante un canale di comunicazione, che è caratterizzato da una *capacita:* il numero massimo di messaggi che è in grado di contenere contemporaneamente. I messaggi quindi vengono posti in una coda, in attesa di essere ricevuti. La lunghezza massima di questa coda, rappresenta la capacità.

A seconda della capacità del canale la *send* può essere sospensiva o no:

- **Canale a capacità nulla:** Se il destinatario non è pronto a ricevere il messaggio, il mittente attende (**Send sincorna)**
- **Canale a capacità non nulla:** Il mittente deposita il messaggio nel canale e continua la sua esecuzione (**Send asincorna).**

Remote Procedure Call_RPC: è un tipo di comunicazione, in cui il mittente si sospende fino a che il destinatario non gli restituisce una risposta. (il messaggio potrebbe richiedere l'esecuzione di un servizio)

5.0 SEGNALI (SINCRONIZZAZIONE TRA I PROCESSI IN UNIX)

La sincronizzazione permette di imporre vincoli sull'ordine di esecuzione delle operazioni dei processi interagenti.
Unix adotta il modello ad ambiente locale, quindi la sincronizzazione può realizzarsi mediante i segnali.

Segnale: è un interruzione software, che notifica un evento asincrono al processo che la riceve.
Un segnale può essere inviato:

- Dal kernel a un processo
- Da un processo utente ad altri processi utente

Quando un processo riceve un segnale può comportarsi in tre modi diversi:

1. Gestire il segnale con una funzione *handler* definita dall'utente
2. Eseguire un'azione predefinita del sistema operativo.
3. Ignorare il segnale.

Nei primi due casi il processo reagisce in modo asincrono al segnale, in altre parole: interrompe la sue esecuzione, esegue un azione (handler, default) ed infine ritorna all'ultima istruzione del codice interrotto.

NB: Non tutti i segnali possono essere gestiti esplicitamente dai processi. Esistono segnali che non sono né intercettabili, né ignorabili. (SIGKILL, SIGSTOP);

5.0.1 SEGNALI UNIX `signal.h`

```
#define SIGHUP  1    /* Hangup(POSIX). Action: exit */
#define SIGINT  2    /* Interrupt (ANSI). Action: exit */
#define SIGQUIT 3    /* Quit(POSIX). Action: exit, core dump*/
#define SIGILL  4    /* Illegal instr.(ANSI).Action: exit,core
dump*/
...
#define SIGKILL 9    /* Kill, unblockable(POSIX). Action: exit*/
#define SIGUSR1 10   /* User-defined signal1 (POSIX). Action:
exit*/
#define SIGSEGV 11   /* Segm. violation(ANSI). Act: exit,core
dump*/
#define SIGUSR2 12   /* User-defined signal2 (POSIX).Act: exit */
#define SIGPIPE 13   /* Brokenpipe (POSIX).Act: exit */
#define SIGALRM 14   /* Alarmclock (POSIX). Act: exit */
#define SIGTERM 15   /* Termination(ANSI). Act:exit*/
...
#define SIGCHLD 17   /* Child status changed(POSIX).Act: ignore*/
#define SIGCONT 18   /* Continue (POSIX).Act. ignore*/
#define SIGSTOP 19   /* Stop, unblockable(POSIX). Act: stop */
```

5.0.2 SYSTEM CALL `signal`

```
void (* signal(int sig, void (*func)()))(int);
```

- **sig**: è l'intero (o il nome simbolico) che individua il segnale da gestire
- **func** è il puntatore ad una funzione che indica l'azione da associare al segnale, il particolare la funzione func può:
    - Puntare alla routine di gestione dell'interruzione (handler)
    - Valere SIG_IGN (nel caso di segnale ignorato)
    - Valere SIG_DFL (nel caso di azione di default)
- Ritorna un puntatore a funzione
    - Al precedente gestore del segnale
    - SIG_ERR(-1) nel caso di errore.

Esempio:

```
#include <signal.h>
void gestore(int);
…
int main(){
…
signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
...
signal(SIGUSR1, SIG_DFL); /*USR1 torna a default */
signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non è ignorabile */
…
}
```

5.0.3 ROUTINE DI GESTIONE DEL SEGNALE (handler)

- L'handler prevede sempre un parametro formale di tipo int che rappresenta il numero del segnale effettivamente ricevuto.
- L'handler non restituisce alcun risultato.

Esempio:

```
/* file segnali1.c */
#include <signal.h>

void handler(int);

int main()
{
      if(signal(SIGUSR1, handler)==SIG_ERR)
            perror("prima signal non riuscita\n");
      if(signal(SIGUSR2, handler)==SIG_ERR)
            perror("seconda signal non riuscita\n");
for (;;);
}

void handler(int signum)
{
      if(signum==SIGUSR1)
            printf("ricevuto sigusr1\n");
      else if(signum==SIGUSR2)
            printf("ricevuto sigusr2\n");
}
```

SIGCHLD è il segnale che il kernel invia a un processo padre quando il figlio termina, è possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione handler per la gestione di SIGCHLD.

Esempio: gestore del SIGCHLD

```
#include <signal.h>
void handler(int);
int main() {
      int PID, i;
      PID=fork();

      if(PID>0) /* padre */
      {
            signal(SIGCHLD,handler);
            for (i=0; i<10000000; i++); /* attività del padre..*/
            exit(0);
      }
      else /* figlio */
      {
            for (i=0; i<1000; i++); /* attività del figlio..*/
            exit(1);
      }
}
```

```
void handler(int signum) {
      int status;
      wait(&status);
      printf("stato figlio:%d\n", status>>8);
}
```

SEGNALI E fork()

Le associazioni segnali-azioni vengono registrate nella User Area del processo.

- Il figlio eredita dal padre le informazioni relative alla gestione dei segnali:
  - Ignora gli stessi segnali ignorati dal padre
  - Gestisce con le stesse funzioni gli stessi segnali gestiti dal padre
  - I segnali a default del figlio sono gli stessi del padre.
    - Successive signal del figlio non hanno effetto sulla gestione dei segnali del padre.

SEGNALI E exec()

Dopo una exec() un processo:

- Ignora gli stessi segnali ignorati prima di exec
- I segnali a default rimangono a default
- I segnali che prima erano gestiti ora vengono riportati a default.

5.1 SYSTEM CALL

```
int kill(int pid, int sig);
```

I processi possono inviare segnali ad altri processi con la kill

- **sig** è l'intero che individua il segnale da gestire (esempio SIGUSR1).
- **pid** specifica il destinatario del segnale:
    - pid>0 l'intero è il pid dell'unico processo destinatario.
    - pid=0 il segnale è spedito a tutti i processi appartenenti al gruppo del mittente
    - pid <-1 il segnale è spedito a tutti i processi con groupId uguale al valore assoluto di pid.
    - pid==-1 vari comportamenti possibili (POSIX non specifica).

```
unsigned int sleep(unsigned int N);
```

Provoca la sospensione del processo per N secondi al Massimo. Se il processo riceve un segnale durante il periodo di sospensione viene risvegliato prematuramente.
Restituisce:

- 0 se la sospensione non è stata interrotta da segnali.
- Se il risveglio è stato causato da un segnale al tempo Ns, sleep restituisce il numero di secondi non utilizzati nell'intervallo di sospensione (N-Ns)

```
unsigned int alarm(unsigned int N);
```

Imposta un timer che dopo N secondi invierà al process oil segnale SIGALRM.
Ritorna:

- 0 se non vi erano time-out impostati in precedenza
- Il numero di secondi mancante allo scadere del time-out precedente.

NB La alarm() non è una funzione sospensiva, l'azione di default associata a SIGALRM è la terminazione.

```
int pause(void);
```

Sospende il processo fino alla ricezione di un qualunque segnale.
Ritorna -1 (errno = EINTR)

6.0 IL FILE SYSTEM

E' quella componente del sistema operativo che fornisce i meccanismi di accesso e memorizzazione delle informazioni (programmi e dati) allocate in memoria di massa.

Realizza i concetti astratti:

- *di file*: unità logica di memorizzazione
- *di direttorio:* insieme di file
- *di partizione:* insieme di file associato ad in particolare dispositivo fisico.

Organizzazione del file system

- *Struttura logica:* presenta alle applicazioni una visione astratta delle informazioni memorizzate, basata su file, directory, partizioni, ecc. Realizza le operazioni di gestione di file e directory: copia, cancellazione, spostamento
- *Accesso:* definisce e realizza i meccanismi per accedere al contenuto del file; in particolare
  - Definisce l'unità di trasferimento da/verso file: **record logico**
  - Realizza i metodi di accesso (sequenziale, casuale, ad indice)
  - Realizza i meccanismi di protezione
- *Organizzazione fisica:* rappresentazione di file e directory sul dispositivo

6.0.1 FILE

Un file è un insieme di informazioni (programmi, dati (in rappresentazione binaria), dati (in rappresentazione testuale)). Ogni file è individuato da (almeno) un nome simbolico ed è **caratterizzato da un insieme di** *attributi.*

- **Tipo**: stabilisce l'appartenenza a una classe (eseguibili, testo, musica, non modificabili, …)
- **Indirizzo**: puntatore/i a memoria secondaria
- **Dimensione**: numero di byte contenuti nel file
- **Data e ora** (di creazione e/o di modifica)

Nei sistemi multiutente sono presenti anche:

- **Utente proprietario**
- **Protezione**: **diritti di accesso** al file per gli utenti del sistema.

Tutti gli attributi sopra elencati sono contenuti all'interno di una struttura dati chiamata **descrittore del file,** ogni descrittore deve essere memorizzato in modo persistente (il sistema operativo mantiene l'insieme di tutti i file presenti nel file system in apposite strutture)

6.0.2 OPERAZIONI SUI FILE

Il compito del sistema operativo è quello di consentire **l'accesso on-line ai file** (ogni volta che un processo modifica un file, tale cambiamento deve essere **immediatamente visibile** a tutti gli altri processi)

Le operazioni più comuni:

- **Creazione**: allocazione di un file in memoria secondaria e inizializzazione dei suoi attributi
- **Lettura** di record logici dal file
- **Scrittura**: inserimento di nuovi record logici all'interno di file
- **Cancellazione**: eliminazione del file dal file system

Per garantire maggior efficienza il sistema operativo mantiene in memoria una struttura che registra i file attualmente in uso (file aperti) **- tabella dei file aperti** ed effettua *memory mapping* (i file aperti vengono temporaneamente trasferiti in memoria centrale >> accessi più veloci) di questi file.

Le operazioni necessarie sono:

- **Apertura**: introduzione di un nuovo elemento nella tabella dei file aperti e eventuale memory mapping del file
- **Chiusura**: salvataggio del file in memoria secondaria ed eliminazione dell'elemento corrispondente dalla tabella dei file aperti

NB: Il proprietario/creatore di un file dovrebbe avere la possibilità di controllare quali azioni sono consentite sul file e soprattutto da parte di chi.

6.0.3 STRUTTURA INTERNA DEI FILE

Ogni dispositivo di memorizzazione secondaria viene partizionato in **blocchi (record fisici**)

Blocco: unità di trasferimento fisico nelle operazioni di I/O da/verso il dispositivo. Sempre di dimensione fissa.

L'utente vede il file come un insieme di **record logici**

Record logico: unità di trasferimento logico nelle operazioni di accesso al file. Sempre di dimensione variabile.

Uno dei compiti del sistema operativo è di stabilire una corrispondenza tra blocchi e record logici.
Di solito la dimensione dei blocchi è superiore a quella dei record (**Impaccamento** di record logici all'interno dei blocchi)

6.0.4 METODI DI ACCESSO

L'accesso ai file può avvenire secondo tre modalità: accesso sequenziale, diretto e a indice. Il metodo di accesso è indipendente, sia dal tipo di dispositivo utilizzato, sia dalla tecnica di allocazione dei blocchi in memoria secondaria.

1. **Accesso sequenziale:** il file è una sequenza di record logici. Per accedere ad un particolare record "R", è necessario accedere prima agli (i-1) record che lo precedono nella sequenza. In questo tipo di accesso è necessario registrare la posizione corrente attraverso un **puntatore al file** che, dopo ogni operazione di accesso si posizionerà sull'elemento successivo a quello letto/scritto.

2. **Accesso diretto:** il file è un insieme di record logici numerati. Molto utile quando si vuole accedere a grossi file per estrarre/modificare poche informazioni, in quanto permette di accedere direttamente ad un particolare record specificandone il numero.

3. **Accesso ad indice:** ad ogni file viene associata una struttura dati contenente l'indice della informazioni contenute. Per accedere ad un record logico si esegue una ricerca nell'indice (utilizzando una **chiave)**

6.0.5 DIRECTORY

La directory (o direttorio) è uno strumento per organizzare i file all'interno del file system. Le operazioni principali che si possono compiere sui direttori sono:

- **Creazione/cancellazione** di directory
- **Aggiunta/cancellazione** di file
- **Listing:** elenco di tutti i file contenuti nella direcotry
- **Ricerca** di file nella direcotry

La struttura logica delle directory può variare a seconda del sistema operativo (a un livello, a due, ad albero, a grafo aciclico...).

- **Struttura a un livello:** un solo directory per ogni file system. Questa struttura presenta un fondamentale problema: in un sistema multiutente, come separare i file dei diversi utenti?

- **Struttura a due livelli:** il primo livello (directory principale): contiene una directory per ogni utente del sistema. Il secondo livello (directory utenti) contengono i file dei rispettivi utenti.

- **Struttura ad albero:** organizzazione gerarchica ad N livelli. Ogni directory può contenere sia file che direttori. Al primo livello c'è un direttorio radice.

- **Struttura a grafo aciclico:** estende la struttura ad albero vista sopra, con la possibilità di inserire link differenti allo stesso file.

6.1 ORGANIZZAZIONE FISICA DEL FILE SYSTEM

Il sistema operativo si occupa anche della realizzazione del file system sui dispositivi di memorizzazione secondaria:

- **Realizzazione dei descrittori** e la loro organizzazione
- **Allocazione dei blocchi fisici**
- **Gestione dello spazio libero**

RICORDA: ogni blocco contiene un insieme di record logici contigui.
Le tecniche più comuni per l'allocazione dei blocchi sul disco sono:

1. **Allocazione contigua:** ogni file è mappato su un insieme di blocchi fisicamente contigui.
    - Vantaggi: basso costo di ricerca di un blocco, possibilità di accesso sequenziale e diretto.
    - Svantaggi: rende difficile individuare lo spazio libero per l'allocazione di nuovi file.
      Aumento dinamico delle dimensione del file.
      Frammentazione esterna: man mano che si riempie il disco, rimangono zone contigue sempre più piccole, a volte inutilizzabili.

2. **Allocazione a lista:** i blocchi sui quali viene mappato ogni file sono organizzati in una **lista concatenata**. I puntatori ai blocchi sono distribuiti sul disco.
    - Vantaggi: non c'è frammentazione esterna. Minor costo di allocazione
    - Svantaggi: lo spazio occupato è maggiore (puntatori), difficoltà di realizzazione dell'accesso diretto. Costo elevato per la ricerca di un blocco.

3. **Allocazione a indice:** a ogni file è associato un blocco(indice) in cui sono contenuti tutti gli indirizzi dei blocchi su cui è allocato il file. Tutti i puntatori ai blocchi utilizzati per l'allocazione di un determinato file sono concentrati in un unico blocco per quel file (**blocco indice**).
    - Vantaggi: gli stessi dell'allocazione a lista con l'aggiunta: possibilità di un accesso diretto e maggiore velocità di accesso.
    - Svantaggi: possibile scarso utilizzo dei blocchi indice.

Riassumendo il concetto di allocazione, gli aspetti caratterizzanti sono:

- **Grado di utilizzo della memoria**
- **Tempo di accesso medio al blocco**
- **Realizzazione dei metodi di accesso**

NB esistono sistemi operativi che adottano più di un metodo di allocazione.

6.3 FILE SYSTEM UNIX

Il File System di Unix è composto da tre tipologie distinte di file:

- **File ordinari**
- **Direttori**
- **Dispositivi fisici** (file speciali contenuti nel direttorio /dev)

Ad ogni file possono essere associati uno o più nomi simbolici, *ma esclusivamente un solo descrittore* detto **i-node**, rappresentato da un intero, l'**i-number**.

6.3.1 ORGANIZZAZIONE FISICA

I file vengono allocati secondo un criterio a indice (a più livelli di indirizzamento).
La formattazione del disco avviene in blocchi fisici.
Il File System è partizionato in 4 regioni:

- **Boot block:** contiene le procedure di inizializzazione del sistema;
- **Super block:** fornisce i limiti delle 4 regioni, il puntatore alla lista dei blocchi liberi e il puntatore alla lista degli i-node liberi.
- **Data blocks:** è l'area in cui memorizzare il file. Contiene i blocchi allocati e quelli liberi.
- **I-list:** contiene la lista di tutti i descrittori dei file (i-node), direttori e dispositivi presenti nel file system, accessibili attraverso l'indice i-number.

L' **i-node** *è il descrittore del file.* Tra i suoi molti attributi vi sono:

- Tipo di file
- Proprietario, gruppo
- Dimensione
- Data
- 12 bit di protezione
- Numero di links
- (13-15) indirizzi di blocchi (a seconda della realizzazione)

6.3.2 INDIRIZZAMENTO

L'allocazione del file non è su blocchi fisicamente contigui; nell'i-node sono contenuti puntatori a blocchi (prendendo il caso in cui ne sono 13) di cui:
- I primi 10 indirizzi riferiscono blocchi di dati (*Indirizzamento diretto*)
- L'11° indirizzo è quello di un blocco contenente a sua volta indirizzi di blocchi dati (1 livello di *indirettezza*)
- Il 12° è a due *livelli di indirettezza*
- Il 13° è a tre *livelli di indirettezza*

1 □ dati 11 □ indirizzi □ dati 12□indirizzi□indirizzi□dati
13□indirizzi□indirizzi□indirizzi□dati

Un blocco ha dimensione 512 byte, contiene 128 indirizzi da 4 byte ciascuno.
Questo comporta che dato un file, 5 kB siano accessibili direttamente, 128 blocchi di dati siano accessibili tramite **indirezione singola** (128*512 byte = 64 kB), 128*128 blocchi dati siano accessibili mediante **indirezione doppia** (128*128*512 byte = 8 MB) e 128*128*128 blocchi lo siano per **indirezione tripla** (128*128*128*512 byte = 1GB).
La *dimensione massima* di un file è quindi = 1GB + 8 MB + 64 kB + 5kB.
Inoltre l'accesso a file di piccole dimensioni è più veloce rispetto al caso di file più grandi.
Protezione: controllo accesso ai file
Ogni file è accessibile secondo tre diverse modalità: **scrittura, lettura, esecuzione**.

Il proprietario può concedere o negare il permesso di accedere al file ad altri utenti. Mentre esiste un utente privilegiato (**root**) che ha accesso incondizionato ad ogni file del sistema.

Bit di protezione
Ad ogni file sono associati 12 **bit di protezione** (nell'i-node) di cui:

- 9 bit (**rwx**) di lettura (**r**ead), scrittura (**w**rite) ed esecuzione (e**x**ecute) per utente proprietario (**U**ser), utenti del gruppo (**G**roup) e tutti gli altri utenti (**O**thers).
- 3 bit di permessi per file eseguibili [Set-User-Id(**SUID**), Set-Group-Id(**SGID**), Save-Text-Image(**sticky**)]

Al processo che esegue un file eseguibile è associato dinamicamente uno User-Id (e Group-Id).
Chi lancia il processo assume temporaneamente *l'identità del proprietario.*
Set-User-Id associa al processo che esegue il file , l'User-Id del proprietario del file;
Group-User-Id associa al processo che esegue il file , il Group-Id del proprietario del file;
Save-Text-Image l'immagine del processo rimane *in area di swap* anche dopo che il processo è terminato, in maniera tale da velocizzare un futuro riavvio.
Per modificare i bit di protezione esiste la System Call i**nt chmod()** e il comando **chmod** in Shell.

In Unix tutte le informazioni relative alla amministrazione del sistema sono rappresentate da file (di root).
Il file **/etc/passwd,** ad esempio, contiene informazioni su utenti, gruppi e password. Esso è accessibile in scrittura solamente dal proprietario; tuttavia ogni utente può accedervi *esclusivamente* per le modifiche relative al proprio username attraverso il comando **/bin/passwd (**di **root).**

6.3.3 IL DIRETTORIO

Anche i direttori sono rappresentati da file nel File System.
Ogni "file-direttorio" contiene un insieme di record logici costituiti da nomerelativo (di ogni file o sottodirettorio del direttorio in esame) e i-number (l'intero che identifica tale file o sottodirettorio).
Ogni record rappresenta un file che appartiene al direttorio.

6.3.4 FILE IN UNIX

L'accesso ai file in Unix è di tipo sequenziale. Un I/O Pointer (ossia un puntatore al file) registra la posizione corrente.
Ad ogni PROCESSO è associata una TABELLA DEI FILE APERTI DI PROCESSO di dimensione limitata (Max 20 el.): ogni elemento rappresenta un file aperto dal processo ed è individuato da un indice intero chiamato FILE DESCRIPTOR. I fd 0,1,2 rappresentano rispettivamente STANDARD INPUT, OUTPUT ed ERROR. La tabella dei file aperti del processo è allocata nella sua USER STRUCTURE.
Per realizzare l'accesso ai file, il SO utilizza due strutture dati globali, allocate nell'area dati del kernel:

- **TABELLA DEI FILE ATTIVI**: per ogni file aperto, contiene una copia del suo i-node (operazioni + efficienti se si vuole ottenere attributi dei file acceduti). Ogni apertura provoca la copia dell'i-node in memoria centrale (se il file non è già in uso);

- **TABELLA DEI FILE APERTI DI SISTEMA**: ha un elemento per ogni operazione di apertura (anche dello stesso file) relativa a file aperti (e non ancora chiusi). Ogni el. contiene l'I/O Pointer (indica la posizione corrente all'interno del file) e un puntatore all'i-node del file nella tabella dei file attivi.

  N.B: se due processi aprono separatamente lo stesso file F, la tabella conterrà due elementi distinti associati a F, quindi due I/O Pointer distinti.
  Inoltre, se un processo padre apre un file e successivamente crea un figlio, l'I/O Pointer viene condiviso anche dal figlio nella TABELLA DEI FILE APERTI DI SISTEMA.

6.4 SYSTEM CALL PER ACCEDERE A FILE

Unix permette ai processi di accedere a file mediante un insieme di system call:

```
int open(char nomefile[], int flag, [int mode]);
```

Restituisce il file descriptor associate al file, o -1 in caso di errore.

- **nomefile[]** = nome del file(assoluto o relativo);
- **flag** = esprime il modo di accesso (**O_RDONLY, O_WRONLY, O_APPEND**), inoltre è possibile aggiungere altri modi (mediante |) come **O_CREAT** (crea il file) e **O_TRUNC** (la lunghezza del file viene troncata a 0);
- **mode** = parametro richiesto soltanto se l'apertura determina la creazione del file (specifica i bit di protezione).

Se la open ha successo viene inserito un elemento (individuato da fd) nella PRIMA posizione libera della TABELLA DEI FILE APERTI DEL PROCESSO, viene inserito un nuovo record nella TABELLA DEI FILE APERTI DI SISTEMA e viene copiato i-node nella TABELLA DEI FILE ATTIVI (se il file non è già in uso).

METODI DI APERTURA (definiti in `<fcntl.h>`)

- O_RDONLY (=0) accesso in lettura
- O_WRONLY (=1) accesso in scrittura
- O_APPEND (=2) accesso in scrittura append.

```
int creat(char nomefile[], int mode);
```

- **nomefile[]** = nome del file(assoluto o relativo).
- **mode** specifica i 12 bit di protezione per il nuovo file.

Il valore restituito dalla creat() è il file descriptor associato al file, o -1 in caso di errore.
Se la creat ha successo, il file viene aperto in modalità scrittura e l'I/O pointer viene posizionato sul primo elemento.

```
int close(int fd);
```

- **fd** è il file descriptor del file da chiudere
- Restituisce l'esito dell'operazione (0 = successo, <0 insuccesso).

Se la close ha successo il file viene memorizzato su disco, viene eliminato l'elemento di indice fd dalla TABELLA DEI FILE APERTI DEL PROCESSO, e vengono eventualmente eliminati (se non condivisi con altri processi) gli elementi corrispondenti dalla TABELLA DEI FILE APERTI DI SISTEMA e dalla TABELLA DEI FILE ATTIVI.

```
int read(int fd, char *buf, int n);
```

- **fd** = file descriptor del file;
- **buf** = area in cui trasferire I byte letti;
- **n** = numero di caratteri da leggere.

In caso di successo **restituisce** un intero positivo (<= n) che rappresenta il numero di caratteri effettivamente letti. Legge quindi n caratteri a partire dall'I/O Pointer, che successivamente viene spostato in avanti di n caratteri (o bytes).
E' previsto un carattere di End-Of-File EOF che marca la fine del file (da tastiera: ^D).

```
int write(int fd, char *buf, int n);
```

- **fd** = file descriptor del file;
- **buf** = area di memoria da cui trasferire I byte scritti;
- **n** = numero di caratteri da scrivere.

In caso di successo restituisce un intero positivo (==n) che rappresenta il numero di caratteri effettivamente scritti. Anche qui, l'I/O Pointer si sposta in avanti di n Bytes.

```
lseek(int fd, int offset, int origine);
```

Funzione usata per spostare l'I/O pointer.

- **fd** = fd del file;
- **offset** = è lo spostamento in byte rispetto all'origine;
- **origine** = può valere:
    - 0: inizio file (**SEEK_SET**)
    - 1: posizione corrente (**SEEK_CUR**)
    - 2: fine file (**SEEK_END**)
- In caso di successo **restituisce** un intero che rappresenta la nuova posizione.

Cancellazione di un file o decrementa il numero dei suoi link

```
int unlink(char *name);
```

- **name** = nome del file;
- **Ritorna** 0 se OK altrimenti -1.

In generale l'effetto della system call unlink è decrementare di 1 il numero di link del file dato (nell'i-node); nel caso in cui il numero dei link risulti 0, allora il file viene cancellato.

Aggiungere un link a un file esistente

```
int link(char *oldname, char *newname);
```

- **oldname** = nome file esistente;
- **newname** = nome associate al nuovo link.

Incrementa il numero dei link associato al file (nell'i-node), aggiorna il direttorio (aggiunta di un nuovo elemento).

Ritorna 0 in caso di successo, -1 se fallisce (*oldname non esiste, newname esiste già oppure oldname e newname appartengono a file system diversi*).

Verificare diritti

```
int access(char *pathname, int amode);
```

- **pathname** = nome file;
- **amode** = esprime il diritto da verificare e può essere:
  - 00 existence (esistenza)
  - 01 execute access (accesso in esecuzione)
  - 02 write access (accesso in scrittura)
  - 04 read access (accesso in lettura).

- **Restituisce** 0 se OK, altrimenti -1.

Leggere attributi

```
int stat(const char *path, struct stat *buf);
```

- **path** = nome file;
- **buf** = è un puntatore a una struttura di tipo STAT nella quale vengono restituiti gli attributi del file (definito nell'header file <sys/stat.h>).

```
struct stat{
dev_t   st_dev;       /* ID of device containingfile */
ino_t   st_ino;       /* i-number*/
mode_t st_mode;       /* protection*/
nlink_t st_nlink;     /* numberof hard links*/
uid_t  st_uid;        /* userID of owner*/
gid_t  st_gid;        /* groupID of owner*/
dev_t  st_rdev;       /* deviceID (ifspecial file) */
off_t  st_size;       /* total size, in bytes*/
blksize_t st_blksize; /* blocksizefor file systemI/O */
blkcnt_t st_blocks;   /* numberof blocks allocated*/
time_t  st_atime;     /* time of last access*/
time_t  st_mtime;     /* time of last modification*/
time_t  st_ctime;     /* time of last status change*/
};
```

- **Ritorna** 0 se OK, altrimenti, in caso di errore, -1.

stat: st_mode
Per interpretare il valore di st_mode, sono disponibili alcune costanti e macro (<sys/stat.h>); ad esempio:

- **S_ISREG(mode)** è un file regolare? (flagS_IFREG)
- **S_ISDIR(mode)** è una directory? (flagS_IFDIR)
- **S_ISCHR(mode)** è un dispositivo a caratteri (file speciale)? (flagS_IFCHR)
- **S_ISBLK(mode)** è un dispositivo a blocchi (file speciale)? (flagS_IFBLK)

Modifica protezione

```
int chmod(char *pathname, char *newmode);
```

- **pathname** = nome file;
- **newmode** = contiene i nuovi diritti.

Modifica proprietario

```
int chown(char *pathname, int owner, int group);
```

- **pathname** = nome file;
- **owner** = uid (user id) del nuovo proprietario;
- **group** = è il gid (group id) del gruppo;

6.5 GESTIONE DEI DIRETTORI

Così come per i file, lettura/scrittura di un direttorio può avvenire soltanto dopo l'operazione di apertura.
Una volta aperto, il direttorio può essere acceduto in:
-lettura (readdir) *da tutti i processi con il diritto di lettura sul direttorio*;
-scrittura *solo il kernel può scrivere sul direttorio*

6.5.1 SYSTEM CALL

Per effettuare un cambio di direttorio

```
int chdir(char *nomedir);
```

- **nomedir** è il nome del direttorio in cui entrare.
- **Restituisce** 0 se OK o -1 se non è andato a buon fine.

Per aprire un direttorio

```
#include <dirent.h>
DIR *opendir(char *nomedir)
```

- **nomedir** è il nome del direttorio da aprire.
- **Restituisce** un puntatore a DIR (tipo di dato astratto predefinito in <dirent.h> che consente di riferire un direttorio aperto) diverso o uguale a NULL in base alla riuscita o meno dell'operazione.

Per chiudere un direttorio

```
#include <dirent.h>
int closedir(DIR dir)
```

- **dir** rappresenta il puntatore che riferisce al direttorio da chiudere.
- **Restituisce** 0 in caso di successo o -1 in caso contrario.

Lettura di un direttorio

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *descr;
descr = readdir (DIR *dir);
```

- **dir** rappresenta il puntatore al direttorio da leggere (valore restituito dalla opendir).
- **Restituisce** un puntatore non nullo in caso di successo, nullo altrimenti.

In caso di successo **readdir** legge un elemento dal direttorio dato e lo memorizza all'indirizzo puntato da descr. **Descr** punta ad una struttura di tipo **dirent** definita nella libreria **<dirent.h>.**

Creazione di un direttorio

```
int mkdir(char *pathname, int mode);
```

- **pathname** è il nome del direttorio da creare
- **mode** esprime i bit di protezione.
- **Restituisce** 0 in caso di successo o un valore negativo in caso contrario.

6.6 COMUNICAZIONE TRA PROCESSI UNIX

Come visto in precedenza i processi Unix non possono condividere memoria (modello ad ambiente locale), l'iterazione tra processi, quindi, può avvenire: mediante le condivisione di file (molto complesso, sincronizzazione dei processi), oppure attraverso specifici strumenti di **Inter Process Communication. (pipe, fifo, socket)**

LA PIPE*:* è un canale unidirezionale (accessibile ad un estremo in lettura ed all'altro in scrittura) con capacità limitata (è in grado di gestire l'accodamento di un numero limitato di messaggi, gestiti in modo FIFO) che permette la comunicazione tra processi.
La pipe rappresenta un chiaro esempio di comunicazione indiretta (Mailbox), in quanto uno stesso processo può sia depositare messaggi nella pipe (mediante il lato scrittura) che prelevarli (mediante il lato di lettura).
NB: la pipe può anche consentire una comunicazione "bidirezionale" tra i processi, ma va rigidamente disciplinata.

6.6.1 SYSTEM CALL PIPE

Creazione di una pipe

```
int pipe(int fd[2]);
```

- **fd** è un puntatore a un vettore di 2 file descriptor, che verranno inizializzati dalla system call in caso di successo:
    - **fd[0]** rappresenta il **lato di lettura** della pipe
    - **fd[1]** è il **lato di scrittura** della pipe.
- La system call **restituisce**: un valore negativo in caso di fallimento; 0 se ha successo.

Se **pipe(fd)** ha successo, vengono allocati due nuovi elementi nella tabella dei file aperti del processo; ed i rispettivi file descriptor vengono assegnati a fd[0] e fd[1].
Ogni lato di accesso alla pipe è visto dal processo in modo omogeneo al file. Si può accedere alla pipe mediante le system call di accesso ai file: **read, write.**
NB: La read realizza la *receive*, mentre la write realizza la *send.*

Sincronizzazione automatica: la pipe, essendo un canale a capacità limitata, richiede la sincronizzazione dei processi. Ma visto che la write e la read da/verso pipe possono essere sospensive, la sincronizzazione avviene automaticamente: in altre parole se la pipe è vuota, un processo che legge si blocca, viceversa, se la pipe è piena, un processo che scrive si blocca.
ATTENZIONE: soltanto i processi appartenenti a una **stessa gerarchia** (cioè, che hanno un antenato in comune) possono scambiarsi messaggi mediante la pipe (processi fratelli, processi padre-figlio, tra nonno e nipote ecc)
CHIUSURA: ogni processo può chiudere un estremo della pipe con una close. Ma un estremo della pipe viene **effettivamente chiuso**, quando tutti i processi che ne avevano visibilità hanno compiuto una close.
Se un processo tenta una lettura da una pipe vuota in cui lato di scrittura è effettivamente chiuso: read torna 0. Mentre se si tenta una scrittura da una pipe il cui lato di lettura è effettivamente chiuso: write tona – 1 (viene inviato un segnale al padre)
SVANTAGGI: la pipe presenta due svantaggi fondamentali: consente la comunicazione solo tra processi che hanno una relazione di parentela e non è persistente: viene distrutta quando terminano tutti i processi che la usano.

6.6.2 SYSTEM CALL DUP

Per duplicare un elemento della tabella dei file aperti di processo

```
int dup(int fd);
```

- **fd** è il file descriptor del file da duplicare

L'effetto di una dup è copiare l'elemento fd nella tabella dei file aperti alla prima posizione libera. Restituisce il nuovo file descriptor, oppure -1 (caso di errore).
Attraverso la dup si può realizzare la **ridirezione di comandi su file** e la **piping** di comandi.

6.7 FIFO

La **PIPE** ha due svantaggi:

- Consente la comunicazione solo tra processi in relazione di parentela.
- Non è persistente: viene distrutta quando terminano tutti i processi che la usano.
- Per ealizzae la comunicazione tra una coppia di processi non appartenenti alla stessa gerarchia? FIFO

FIFO
E' una pipe con nome nel file system:

- Canale unidirezionale del tipo first-in-first-out.
- È rappresentata da un file system: **persistenza, visibilità** potenzialmente globale.
- Ha un proprietario, un insieme di diritti ed una lunghezza.
- È creata dalla system call mkfifo.
- È aperta e acceduta con le stesse system call dei file.

Creazione di una FIFO

```
int mkfifo(char* pathname, int mode);
```

- **pathname** è il nome della fifo.
- **mode** esprime i permessi.
- **Restituisce** 0 in caso di successo, un valore negativo.

Apertura/chiusura di fifo
Una volta ceata una fifo può essere aperta (come tutti i file), mediante una open; ad esempio, un processo destinataio di messaggi:

```
int fd;
fd=open("myfifo", O_RDONLY);
```

Per chiudere una fifo si usa la close:

```
close(fd);
```

Per eliminare una fifo, si usa la unlink:

```
unlink("myfifo");
```

Accesso a fifo
Una volta aperta la fifo può essere acceduta (come tutti i file), mediante read/write; ad esempio, un processo destinatario di messaggi:

```
int fd;
char msg[10];
fd=oepn("myfifo", O_RDONLY);
read(fd, msg, 10);
```

7.0 MULTIPROGRAMMAZIONE E GESTIONE DELLA MEMORIA

L'obiettivo principale della multiprogrammazione è l'uso efficiente delle risorse computazionali (efficienza nell'uso della CPU, velocità di risposta dei processi...) e la necessità di mantenere più processi in memoria centrale, gestendo la memoria in modo da consentire la presenza contemporanea di più processi. A livello hardware ogni sistema ha un unico spazio di memoria accessibile direttamente da CPU e dispositivi.
Compiti del Sistema Operativo sono:

1. Accesso alla memoria centrale (vettore di celle, ognuna univocamente individuata da un indirizzo): deve svolgere load e store di dati e istruzioni. Gli indirizzi possono essere simbolici (riferimenti a celle di memoria nei programmi in forma sorgente mediante nomi simbolici), logici (riferimenti a celle nello spazio logico di indirizzamento) e fisici(riferimenti assoluti a livello hardware). Ogni processo dispone di un proprio spazio di indirizzamento logico che viene allocato nella memoria fisica.

2. Binding degli indirizzi, ossia l'associazione ad ogni indirizzo logico o simbolico(relativo) di un indirizzo fisico(assoluto). Può essere svolto - staticamente, che a sua volta può essere a tempo di compilazione (in cui il compilatore genera indirizzi assoluti) o a tempo di caricamento (in cui il compilatore genera degli indirizzi relativi che saranno poi convertiti dal loader in indirizzi assoluti). - dinamicamente, a tempo di esecuzione (in cui un processo può essere spostato da un'area a un'altra durante l'esecuzione).

3. Caricamento/collegamento dinamico, il cui obiettivo è l'ottimizzazione della memoria. Per il caricamento dinamico è possibile usare un loader di collegamento rilocabile che carica e collega dinamicamente la funzione al programma che la usa. Le funzioni possono essere usate da più processi simultaneamente, cosa che deve essere gestita dal SO gestendo gli accessi dei processi allo spazio di altri processi o l'accesso di più processi agli stessi indirizzi.

4. Gestire spazi di indirizzi logici di dim. maggiore allo spazio fisico, l'overlay è la soluzione al problema in cui la memoria disponibile non è sufficiente ad accogliere codice e dati di un processo. L'overlay mantiene in memoria istruzioni e dati più frequenti e che sono necessari nella fase corrente. Codice e dati di un processo vengono suddivisi in overlay che vengono caricati e scaricati dinamicamente dal gestore di overlay.

7.0.1 TECNICHE DI ALLOCAZIONE DELLA MEMORIA CENTRALE

Codice e dati dei processi possono essere allocati in memoria centrale con due tipi di allocazione: contigua e non contigua.
Contigua a partizione singola: la parte di memoria per l'allocazione dei processi non è partizionata e solo un processo alla volta può essere allocato in memoria (no multiprogrammazione).
Contigua a partizioni multiple: c'è multiprogrammazione e quindi necessità di proteggere codice e dati di ogni processo. Ad ogni processo viene associata una partizione, cioè un'area di memoria distinta.
Le partizioni possono essere fisse o variabili:

- **Fisse** (MFT, Multiprogramming with fixed number of tasks): la dimensione di ogni partizione viene fissata a priori e per ogni processo viene cercata una partizione libera di sufficiente dimensione. Problemi possono essere: la presenza di una

frammentazione interna dovuta al sottoutilizzo della partizione, un grado di multiprogrammazione limitato al numero delle partizioni e inoltre la limitazione della dimensione massima dello spazio di indirizzamento di un processo alla dimensione della partizione più estesa.

- **Variabili** (MVT, Multiprogramming with variable number of tasks): ogni partizione viene allocata dinamicamente e dimensionata in base alla dimensione del processo da allocare e quando un processo viene schedulato il sistema operativo cerca un'area sufficientemente grande dove allocare dinamicamente la partizione associata. Viene eliminata la frammentazione interna, in quanto ogni partizione è grande quanto la dimensione del processo, il grado di multiprogrammazione limitato viene sostituito da un grado variabile e la dimensione massima dello spazio di indirizzamento è limitato dalla dimensione dello spazio fisico e non più dalla dimensione della partizione più estesa. Gli unici problemi sono la scelta dell'area migliore in cui allocare e la frammentazione esterna, in quanto più si allocano nuove partizioni più la memoria libera è sempre più frammentata e c'è necessità di compattazione.

Per eliminare la frammentazione esterna si passa all'allocazione non contigua, che si divide in paginazione e segmentazione. Questo concetto si basa sulla partizione dello spazio fisico di memoria in pagine(frame) di dimensione costante, sulle quali mappare porzioni dei processi da allocare. Con la paginazione lo spazio fisico viene considerato come un insieme di frame (slot liberi da riempire) di dim Df costante prefissate e lo spazio logico come un insieme di pagine(slot pieni) di dimensione uguale a Df. Ogni pagina logica di un processo caricato in memoria viene mappata su una pagina fisica in memoria centrale. In questo modo non si parla più di frammentazione esterna ed è possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo.

Indirizzo logico: p+d; p=numero di pagina logica, d=offset della cella rispetto all'inizio della pagina. Sono indirizzi di m bit (n bit dell'offset e m-n della pagina): dim max dello spazio logico=2^m, dim della pagina=2^n, num di pagine=2^m-n.
Indirizzo fisico: f+d; f=numero di frame (pagina fisica), d=offset della cella rispetto all'inizio del frame.

Il binding fra indirizzi logici e fisici può essere realizzato mediante tabella della pagine, che a ogni pagina logica associa la pagina fisica corrispondente.
Dato che la tabella può essere molto grande e la traduzione della corrispondenza tra i due indirizzi deve essere molto veloce ci sono varie soluzioni:

1. **Su registri CPU** (accesso veloce, cambio di contesto pesante, dimensioni limitate della tabella);
2. **In memoria centrale** (registro apposito PageTableBaseRegister che memorizza la collocazione della tabella, 2 accessi in memoria per ogni operazione di load o store);
3. **Memoria centrale + cache** (nella cache è presente la Translation Look-aside Buffers(TLB) per velocizzare l'accesso, in questo modo se la coppia(pagina,frame) è già presente in TLB l'accesso è veloce, altrimenti si va in memoria centrale). Il TLB è inizialmente vuoto, poi quando si accede alle pagine viene riempito. L'Hit-Ratio è la percentuale di volte che una pagina viene trovata in TLB. La tabella delle pagine ha dimensione fissa e ad ogni elemento viene associato un bit che se è a 1 significa che la pagina appartiene allo spazio logico del processo, se è a 0 l'entry non è valida. E' anche presente un registro, **Page Table Length Register**, che contiene il numero degli

elementi validi nella tabella delle pagine. Inoltre per ogni entry della tabella delle pagine è possibile trovare dei bit di protezione che indicano la modalità di accesso alla pagina.

Quando lo spazio logico di indirizzamento di un processo è molto esteso si ha un elevato numero di pagine e una tabella delle pagine di grandi dimensioni. In questo caso si può utilizzare la paginazione a più livelli, che consiste nell'allocazione non contigua anche della tabella delle pagine, ossia si applica la paginazione alla tabella delle pagine. In questo modo è possibile indirizzare spazi logici di dimensioni elevate e mantenere in memoria solo le pagine che servono. L'unico problema è il tempo di accesso più elevato dovuto ad un maggior numero di accessi in memoria.
Per limitare l'occupazione della memoria in alcuni SO si usa un'unica struttura dati globale, la tabella delle pagine invertita, che ha un elemento per ogni frame. Ogni elemento rappresenta un frame e se è allocato contiene: un pid che identifica il processo a cui è assegnato il frame, un p che identifica il numero di pagina logica e un d che è l'offset all'interno della pagina. Preso un indirizzo logico si cerca nella tabella l'elemento che contiene la coppia (pid,p) e l'indice dell'elemento trovato rappresenta il numero del frame allocato alla pagina logica p. Il tempo di ricerca è più alto e c'è un problema di condivisione di codice tra processi (rientranza) in quanto è difficile associare un frame a più pagine logiche di processi diversi.

7.1.0 SEGMENTAZIONE

La segmentazione si basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti(segmenti), caratterizzate da nome e lunghezza. Non è stabilito un ordine tra i segmenti, c'è una divisione semantica per funzione (es: -codice-stack-dati-heap), ogni segmento viene allocato in memoria in modo contiguo e ad ognuno di essi il SO associa un intero attraverso il quale lo si può riferire.
Nella segmentazione ogni indirizzo logico ha la struttura

< SEGMENTO – OFFSET >

Dove:

- **Segmento**: numero che individua il segmento del sistema.
- **Offset**: posizione della cella del segmento.

E' presente una tabella dei segmenti che ha una entry per ogni segmento che ne descrive l'allocazione in memoria fisica mediante la coppia:

< BASE – LIMITE >

Dove:

- Base: indirizzo della prima cella del segmento nello spazio fisico.
- Limite: dimensione del segmento.

Questa tabella può avere dimensioni elevate e può essere realizzata su registri di CPU, in memoria, con registri base (STBR) e limite (STLR) oppure su cache, che conterrà i segmenti recenti.
La segmentazione (più segmenti per processo) è l'evoluzione della tecnica di allocazione a partizioni variabili (1 segmento per processo). Il problema principale è la frammentazione esterna, che viene risolta con l'allocazione dei segmenti con politiche adatte alla situazione (best fit, worst fit, ..).
La segmentazione e la paginazione possono essere unite nella segmentazione paginata, in cui lo spazio logico è segmentato e ogni segmento è suddiviso in pagine, con questo metodo si elimina la frammentazione esterna e non è necessario mantenere in memoria l'intero

segmento, ma basta caricare solo le pagine necessarie (vedi memoria virtuale). Sono presenti una tabella dei segmenti e una tabella delle pagine per ogni segmento.

7.1.1 MEMORIA VIRTUALE

La dimensione della memoria può rappresentare un vincolo importante riguardo alla dimensione dei processi e al grado di multiprogrammazione. Si può quindi volere un sistema di gestione della memoria che consenta la presenza di più processi in memoria, indipendentemente dalla dimensione dello spazio disponibile e una memoria che svincoli il grado di multiprogrammazione dalla dimensione effettiva della memoria. La memoria virtuale, a differenza delle tecniche precedenti in cui tutto lo spazio logico di ogni processo è allocato in memoria o in cui era presente l'overlay e il caricamento dinamico, permette l'esecuzione di processi non completamente allocati in memoria. Questa tecnica ha molti vantaggi: innanzitutto la dimensione dello spazio logico degli indirizzi non è vincolata dall'estensione della memoria, il grado di multiprogrammazione è indipendente dalla dimensione della memoria fisica, il caricamento di un processo e swapping (vedi dopo) hanno un costo ridotto e il programmatore non deve preoccuparsi dei vincoli relativi alla dimensione della memoria.
Di solito la memoria virtuale è realizzata mediante tecniche di paginazione su richiesta in cui tutte le pagine di ogni processo risiedono in memoria di massa (backing store) e durante l'esecuzione alcune di esse vengono trasferite all'occorrenza in memoria centrale, è infatti presente un pager che è un modulo del SO che realizza i trasferimenti delle pagine (*swapper* di pagine).
Pager lazy("pigro"): trasferisce in memoria centrale una sola pagina se ritenuta necessaria. Se si utilizza la paginazione su richiesta, l'esecuzione di un processo può richiedere swap-in del processo. In questo caso è necessaria la presenza di uno swapper che gestisce i trasferimenti di interi processi (mem centrale -> mem secondaria) e di un pager che gestisce i trasferimenti di singole pagine. Quest'ultimo prima di eseguire swap-in di un processo può prevedere le pagine di cui il processo avrà bisogno nella fase di caricamento. E' presente una tabella delle pagine e una memoria secondaria, con strutture necessarie per la sua gestione. Una pagina dello spazio logico di un processo può essere allocata in memoria centrale o secondaria e lo si distingue dai bit di validità presenti nella tabella delle pagine; la pagina può anche essere invalida e in quel caso non esiste lo spazio logico del processo -> interruzione al SO (page fault).
Quando viene ricevuta questa interruzione avviene:

1. Salvataggio del contesto di esecuzione del processo(resgistri, stato, tabella delle pagine).
2. Verifica del motivo del page fault: mediante una tabella interna al kernel si verifica se il processo, in cui è presente la pagina, non ha realmente spazio logico(riferimento illegale) ->terminazione del processo, o se la pagina è in memoria secondaria(riferimento legale).
3. Copia della pagina in un frame libero.
4. Aggiornamento della tabella delle pagine.
5. Ripristino del processo ed esecuzione dell'istruzione interrotta.

In seguito a un page fault se è necessario caricare una pagina in memoria centrale potrebbero non esserci frame liberi. In questo caso si utilizza la sovrallocazione: viene sostituita una pagina vittima Pvitt allocata in memoria con la pagina da caricare Pnew. Per prima cosa si individua la Pvitt, la si salva su disco, si carica la Pnew nel frame liberato, si aggiornano le tabelle e si riprende il processo. Durante questo procedimento, la sostituzione di una pagina può richiedere 2 trasferimenti da/verso il disco (scaricare la vittima e caricare la pagina

nuova). E' possibile che la vittima in memoria non sia stata modificata dalla sua copia presente su disco (es: read-only) e in quel caso si può semplicemente cancellare la vittima sul disco senza copiarla sul disco, in quanto uguale. Viene così inserito in ogni elemento della tabella delle pagine un bit di modifica (dirty bit). Se è a 1 la pagina ha subito almeno un aggiornamento da quando è caricato in memoria, se invece è a 0 significa che la pagina non è stata modificata. L'algoritmo di sostituzione esamina il bit di modifica della vittima ed esegue swap-out della vittima solo se il dirt-bit è settato.

7.1.2 ALGORITMI DI SOSTITUZIONE

Il fine di ogni algoritmo è di sostituire quelle pagine la cui probabilità di accesso a breve termine è bassa.
Algoritmi:

- **LFU (Least Frequently Used)** sostituita la pagina che è stata usata meno frequentemente (in un intervallo di tempo prefissato); in questo caso è necessario associare un contatore d'accessi ad ogni pagina.
- **FIFO** sostituita la pagina che è da più tempo caricata in memoria (indipendentemente dal suo uso); in questo caso è necessario memorizzare la cronologia dei caricamenti in memoria.
- **LRU (Least Recently Used)** sostituita la pagina che è stata usata meno recentemente; in questo caso è necessario registrare la sequenza degli accessi alle pagine in memoria. Questo si può fare con il time stamping (l'elemento della tabella delle pagine contiene un campo che rappresenta l'istante dell'ultimo accesso alla pagina) o con lo Stack (ogni elemento rappresenta una pagina e l'accesso a una pagina provoca lo spostamento dell'elemento corrispondente al top dello stack). Spesso vengono usate versioni semplificate di LRU, inserendo un bit di uso associato alla pagina che al momento del caricamento è inizializzato a 0 e quando la pagina viene acceduta, viene settato. Periodicamente i bit di uso vengono resettati. Verrà sostituita una pagina avente bit di uso uguale a 0. Il criterio potrebbe inoltre considerare il dirty bit: se infatti ci fossero più pagine non usate di recente (cioè con bit di uso uguale a 0), ne verrebbe scelta una non aggiornata (cioè con dirty bit uguale a 0).

A volte c'è la possibilità che il processo impieghi più tempo per la paginazione che per l'esecuzione: in questo caso si parla di trashing. Per contrastarlo si usano tecniche di gestione della memoria che si basano su pre-paginazione, si prevede quindi il set di pagine di cui il processo da caricare ha bisogno per la prossima fase di esecuzione: il working set. Il working set può essere individuato in base a criteri di località temporale (vedi dopo).

7.1.3 LOCALITA' DI PROGRAMMI

Un processo in una certa fase di esecuzione usa solo un sottoinsieme relativamente piccolo delle sue pagine logiche, che varia lentamente nel tempo.

- **Località spaziale:** alta probabilità di accedere a locazioni vicine nello spazio logico/virtuale) a locazioni appena accedute.
- **Località temporale**: alta probabilità di accesso a locazioni accedute di recente.]

Dato un intero d, il working set di un processo P nell'istante t è l'insieme di pagine d(t) indirizzate da P nei più recenti d riferimenti, d definisce la "finestra" del working set, d caratterizza il working set, esprimendo l'estensione della finestra dei riferimenti. Se d è piccolo il working set è insufficiente a garantire località (alto numero di page fault), se d è grande: allocazione di pagine non necessarie.
Ad ogni istante, data la dim corrente del working set WSSi di ogni processo Pi, si può individuare D=Ë(sommatoria) WSSi, richiesta totale di frame; e se m è il numero totale di

frame liberi può esserci spazio per l'allocazione di nuovi processi se D<m o swapping di uno (o più) processi se D>m. In assenza di memoria virtuale, swapper ricopre un ruolo chiave per la gestione delle contese di memoria da parte dei diversi processi. Infatti lo swapper periodicamente viene attivato per provvedere eventualmente a swap-in (processi piccoli e processi da più tempo swapped) e swap-out (processi inattivi, processi da più tempo in memoria) di processi.

7.2.0 PREPAGINAZIONE CON WORKING SET

Quando si carica un processo, si carica un working set iniziale, che verrà aggiornato dinamicamente, in base al principio di località temporale: all'istante t vengono mantenute le pagine usate dal processo nell'ultima finestra d(t) e le altre pagine esterne possono essere sostituite, in questo modo si riduce il numero di page fault.
In UNIX si utilizza la segmentazione paginata, la memoria virtuale tramite paginazione su richiesta senza working set e l'allocazione di ogni segmento non è contigua. Si utilizza pre-paginazione ed è presente una core-map, ossia una struttura dati interna al kernel che descrive lo stato di allocazione dei frame e che viene consultata in caso di page fault. Si usano bit di uso e dirty bit e la sostituzione della pagina viene eseguita dal pager pagedaemon (pid=2). La sostituzione delle pagine viene attivata dallo scheduler quando il numero totale dei frame liberi è ritenuto insufficiente, ossia quando è minore del valore di lotsfree (numero minimo di frame liberi per evitare sostituzione di pagine). Sono presenti anche i parametri minfree (numero minimo di frame liberi necessari per evitare swapping dei processi) e desfree (numero desiderato di frame liberi).
Lo scheduler attiva swapper se il sistema di paginazione è sovraccarico, ossia il numero di frame liberi è minore del minfree e il numero medio di frame liberi nell'unità di tempo è minore del destre. Lo spazio di indirizzamento di ogni processo può essere suddiviso in un insieme di regioni omogenee e contigue, costituite da una sequenza di pagine accomunate dalle stesse caratteristiche di protezione e paginazione. Ogni pagina ha dimensione costante.

8.0 PROGRAMMAZIONE CONCORRENTE NEL MODELLO AD AMBIENTE LOCALE

In una macchina ad ambiente globale le comunicazioni attraverso dati e risorse condivise devono essere sincronizzate negli accessi. Esistono diverse tecniche di risoluzione gestite dal NUCLEO come semafori, regioni critiche e monitor.

Ogni applicazione può essere rappresentata come un insieme di due componenti disgiunti:

- **PROCESSI** (componenti attivi)
- **RISORSE** (componenti passivi) è un oggetto fisico o logico che viene utilizzato dal processo per la suo completamento.
  Le risorse sono raggruppate in CLASSI: ogni classe identifica l'insieme di tutte le sole operazioni che un processo può eseguire per operare su risorse di quella classe.
  - **Risorsa PRIVATA** (di un processo P): P è il solo processo che può eseguire operazioni su quella risorsa.
  - **Risorsa COMUNE** (o globale): più processi possono operare sulla risorsa. Nel modello ad ambiente locale i processi interagiscono solamente operando su risorse comuni.

  Un gestore di risorsa (generalmente è il programmatore) è un'entità in grado di definire ogni istante t l'insieme dei processi che possono accedere alla risorsa R:
  - **Risorsa DEDICATA**: se il gestore (SR(t)) contiene al più un processo.
  - **Risorsa CONDIVISA**: se SR(t) contiene più processi.

  Altra suddivisione riguarda l'allocazione:
  - **Risorsa allocata STATICAMENTE**: il gestore definisce l'insieme dei processi all'istante T0 senza modificarlo durante l'elaborazione.
  - **Risorsa allocata DINAMICAMENTE**: l'insieme SR varia nel tempo. Generalmente SR è inizialmente vuoto ed ogni processo che vuole utilizzare quella risorsa chiede al gestore il permesso, che può accettare, rifiutare o ritardare la richiesta.

8.0.1 IL PROBLEMA DEL DEADLOCK:

Un insieme di processi è in deadlock se ogni processo dell'insieme è in attesa di un evento che può essere causato solo da un altro processo dell'insieme (si instaura uno stallo).
CONDIZIONI NECESSARIE PER IL BLOCCO CRITICO:

1. **MUTUA ESCLUSIONE**: le risorse sono utilizzate in modo mutuamente esclusivo.
2. **POSSESSO E ATTESA**: ogni processo che possiede una risorsa può richiederne un'altra.
3. **IMPOSSIBILITA' DI PRELAZIONE**: una volta assegnata ad un processo, una risorsa non può essere sottratta al processo (no preemption).
4. **ATTESA CIRCOLARE**: esiste un gruppo di processi {P0, P1, … , PN} in cui P0 attende una risorsa posseduta da P1, P1 attende una risorsa posseduta da P2, … e PN attende una risorsa posseduta da P0.

Se almeno UNA delle QUATTRO condizioni non è verificata NON C'E' DEADLOCK.

Le situazioni di deadlock devono essere evitate o prevenendo il blocco critico o curandolo:

- **PREVENZIONE**: può essere STATICA (si impongono dei vincoli a priori che evitano almeno una delle 4 situazioni sopra citate) o DINAMICA (prima che un processo P richieda la risorsa R, si controlla se c'è la possibilità di deadlock).
- **RILEVAZIONE / RIPRISTINO DEL DEADLOCK**: non c'è prevenzione, ma il S.O. rileva la presenza del deadlock e attua un algoritmo di ripristino (il più conosciuto è l'ALGORITMO DEL BANCHIERE) (vedi slide).

Per la prevenzione statica si possono seguire le seguenti tracce:

1. **MUTUA ESCLUSIONE**: si utilizzano risorse condivisibili (in certi casi non è possibile);
2. **POSSESSO E ATTESA**: si impedisce ad ogni processo di possedere una risorsa mentre ne richiede un'altra;
3. **PREEMPTION**: possibilità di sottrarre la risorsa al processo (in certi casi non è possibile);
4. **ATTESA CIRCOLARE**: si stabilisce un rigido ordinamento nell'acquisizione delle risorse da parte di ogni processo.

8.0.2 MUTUA ESCLUSIONE

Il problema della mutua esclusione nasce quando più di un processo alla volta può avere accesso a variabili / risorse comuni. La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo.

Un'istruzione I(d) che opera su un dato d è detta **INDIVISIBILE** (o atomica) se durante la sua esecuzione da parte di un processo P, il dato d non è accessibile ad altri processi. La sequenza di istruzioni con le quali un processo accede e modifica un insieme di **VARIABILI COMUNI** prende il nome di **SEZIONE CRITICA**.

La **REGOLA DI MUTUA ESCLUSIONE** stabilisce che una sola sezione critica di una classe (insieme di sezioni critiche) può essere in esecuzione ad ogni istante.

I requisiti per risolvere i problemi della mutua esclusione sono:

1. Sezioni critiche della stessa classe eseguite in modo esclusivo;
2. Quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione ad altri processi;
3. Assenza di deadlock.

Lo schema generale di risoluzione è assicurare un PROLOGO ed un EPILOGO all'inizio e alla fine della sezione critica:

- **PROLOGO**: ogni processo prima di entrare in una sezione critica deve chiedere l'autorizzazione che gli garantiscono l'uso ESCLUSIVO della risorsa, se questa è libera, oppure ne impediscono l'accesso se questa è già occupata.
- **EPILOGO**: al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la sezione critica.

Tra le tante soluzioni adottate *l'algoritmo di Dekker* è **STARVATION-FREE**.
Con il termine *STARVATION* si intende un fenomeno che si verifica quando un processo attende un tempo INFINITO per l'acquisizione di una risorsa, mentre per *STARVATION-FREE* si intende un algoritmo che garantisce ad ogni processo di accedere alla sezione critica in un tempo FINITO.

Ai requisiti sopra citati aggiungiamo quindi:

a. Assenza di **STARVATION**;
b. Eliminare **l'ATTESA ATTIVA** (ossia sospendere l'esecuzione di un processo per tutto il tempo in cui non può avere accesso alla sezione critica.

ES: nelle operazioni di Lock / Unlock sono rispettati i requisiti 1,2,3, il 4 non è implicito in quanto occorre realizzare un meccanismo di arbitraggio per l'accesso in memoria, mentre il 5 non è soddisfatto perché nella lock è presente una forma di attesa attiva.

9.0 THREAD IN JAVA

THREAD: flusso sequenziale di controllo all'interno di un processo. *Condivide codice, dati e spazio di indirizzamento con gli altri thread associati*. Ha stack e program counter privati.
Ad ogni programma Java corrisponde l'esecuzione di un task (processo), contenente almeno un singolo thread, corrispondente al metodo main() sulla JVM (java virtual machine).
È comunque possibile creare dinamicamente thread che eseguono concorrentemente all'interno del programma.
Essi sono implementabili secondo due modalità distinte:

1) Estendendo la classe Thread;
2) Implementando l'interfaccia Runnable.

Gli oggetti thread derivano dalla classe Thread contenuta nel package **Java.lang**. Il metodo **run()** di tale classe definisce le istruzioni che ogni thread (oggetto della classe) andrà ad eseguire. Questo metodo, la cui implementazione è vuota nella classe Thread, va ridefinito in ogni sottoclasse derivata, andando a specificare le modalità di esecuzione di ogni thread.
Per creare un thread bisogna creare una nuova istanza della classe che lo definisce tramite **new**, dopodiché per attivarlo, nel main, bisogna richiamare il metodo **start()** che a sua volta invoca run().

Il ciclo di vita del thread può essere schematizzato in 4 fasi:

- **New Thread** subito dopo la creazione tramite new;
- **Runnable** il thread è eseguibile, ma potrebbe non essere in esecuzione;
- **Not Runnable** stato nel quale un thread finisce quando non può essere messo in esecuzione, a causa dell'invocazione di metodi quali wait(), suspend(), sleep(), o perché in attesa della terminazione di un'operazione di I/O. Ne esce nel momento in cui vengono richiamati metodi appositi.
- **Dead** stato finale in cui giunge il thread per "morte naturale" o perché un altro thread ha invocato il suo metodo **stop().**

La classe che implementa Runnable deve ridefinire il metodo run(); viene poi creata una istanza di tale classe tramite new e a questo punto si crea sempre tramite new una nuova istanza della classe Thread, a cui va passato come parametro l'oggetto che implementa Runnable. Infine si esegue il thread invocando, tramite l'oggetto Thread creato, il metodo start().

Dal momento che thread appartenenti allo stesso task condividono lo stesso spazio di memoria (heap) occorrono meccanismi di sincronizzazione che assicurino la corretta esecuzione.
Le interazioni tra thread avvengono mediante oggetti comuni, distinguiamo tra interazioni:
- Di tipo **competitivo**: Mutua esclusione (meccanismo Object locks)
- Di tipo **cooperativo**: Semafori, Variabili condizioni (meccanismo wait - notify)

9.0.1 MUTUA ESCLUSIONE

La JVM associa automaticamente ad ogni oggetto un **lock** rappresentante lo stato: libero/occupato. Inoltre esiste la parola chiave **synchronized** per identificare alcune porzioni di codice dette "sezioni critiche"; mediante essa il compilatore fa sì che all'inizio della sezione critica venga acquisito il lock, e al termine di essa venga rilasciato.
Entry set = coda nella quale viene sospeso il thread nel caso in cui il lock ad esso associato risulti bloccato.

9.0.2 WAIT E NOTIFY

Wait set: coda di thread associata ad ogni oggetto, inizialmente vuota.
I thread entrano ed escono dal **wait set** utilizzando i metodi **wait()** e **notify().**
Wait e notify possono essere invocati da un thread solo all'interno di un blocco sincronizzato o di un metodo sincronizzato (è necessario il possesso del lock dell'oggetto).

- **wait()**: comporta il rilascio del lock, la sospensione del thread ed il suo inserimento in wait set. Al risveglio (tramite notify o notifyall) il thread riacquisirà automaticamente il lock.
  (NB. ThrowsInterruptedException).
- **notify()**: comporta l'estrazione di un thread da wait set ed il suo inserimento in entry set. (Cioè: il thread risvegliato riacquisisce automaticamente il lock).
- **notifyall()**: comporta l'estrazione di tutti i thread da wait set ed il loro inserimento in entry set.

Il limite di una sincronizzazione mediante questi metodi sta nel fatto che per ogni oggetto esiste un'unica coda (wait set) su cui poter sospendere i thread. Quindi per esempio nel sistema "produttore/consumatore" dopo ogni prelievo e inserimento bisogna risvegliare tutti i processi sospesi.

9.1.0 SEMAFORI

Un semaforo è uno strumento di sincronizzazione che consente di risolvere qualunque problema di sincronizzazione tra thread nel modello ad ambiente globale. Generalmente è realizzato dal nucleo del sistema operativo. Un semaforo è un dato astratto rappresentato da un intero NON NEGATIVO a cui è possibile accedere solo tramite le due operazioni P e V:

- P(s): ritarda il processo fino a che il valore del semaforo diventa maggiore di 0 e quindi decrementa tale valore di 1;
- V(s): incrementa di 1 il valore del semaforo.

Le due operazioni sono ATOMICHE (utilizziamo LOCK / UNLOCK). Il valore del semaforo viene modificato da un solo processo alla volta. Funzionamento:

- S=0: l'esecuzione di P provocherà l'attesa del processo che la invoca (ROSSO);
- S=1: la chiamata di P provocherà il decremento di S e la continuazione dell'esecuzione (VERDE).

Ad ogni semaforo, inoltre, è associata una coda Qs nella quale sono posti i descrittori dei processi che attendono l'autorizzazione a procedere.

La costruzione di semafori fino alla versione 5.0 di Java era prevista tramite wait e notify. Dalla versione 6 è disponibile la classe **java.util.concurrent.Semaphore**, tramite la quale possiamo creare semafori ed utilizzarli medianti i metodi **acquire()** (che di fatto sarebbe l'implementazione di p()) e **release()** (l'implementazione di v()).

Alcuni metodi dei thread quali **stop()** e **suspend()** possono creare alcuni problemi.

- **stop()** forza la terminazione di un thread liberando immediatamente le risorse da esso utilizzate, nel caso in cui il thread in esame stava effettuando operazioni da svolgersi in maniera *atomica*, l'interruzione può condurre ad *uno stato inconsistente del sistema.*
- **suspend()** blocca un thread in attesa di una successiva invocazione di resume(), senza liberare le risorse che esso utilizza; se tale thread aveva acquisito una risorsa in maniera esclusiva, essa resta bloccata.

RIASSUMENDO:
Utilizzando i semafori, la decisione se un processo possa proseguire l'esecuzione dipende dal valore di un solo semaforo e la scelta del processo da risvegliare avviene tramite l'algoritmo implementato nella V (**FIFO**).

In problemi di sincronizzazioni più complessi, invece, la decisione se un processo possa proseguire l'esecuzione dipende in generale dal verificarsi di una *CONDIZIONE DI SINCRONIZZAZIONE*, mentre la scelta del processo da risvegliare può avvenire in base a criteri diversi da quelli adottati nell'implementazione di V (ES: sulla base di priorità tra processi).

Per ovviare a questi problemi si utilizzano costrutti linguistici di "più alto livello" come ad esempio il MONITOR.

9.2.0 MONITOR

E' un costrutto sintattico che associa un insieme di operazioni (public o entry) ad una struttura dati comune a più processi, tale che le operazioni public siano le sole operazioni permesse su quella struttura e siano mutualmente esclusive (sono le uniche operazioni che consentono di modificare le variabili locali).

Le variabili locali sono accessibili solo all'interno del monitor, mentre le operazioni dichiarate non public non sono accessibili dall'esterno.

Ad ogni monitor è associata una risorsa: lo scopo del monitor è controllarne gli accessi in accordo a determinate politiche. L'accesso avviene mediante DUE LIVELLI DI SINCRONIZZAZIONE:

1. Il primo garantisce che solo un processo alla volta possa aver acceso alle variabili comuni del monitor (realizzato direttamente dal linguaggio → metodi public);
2. Il secondo controlla l'ordine con il quale i processi hanno accesso alla risorsa in base ad una condizione di sincronizzazione che assicura l'ordinamento (posto in attesa se la condizione è falsa): viene realizzato dal programmatore in base alle politiche di accesso date → si usa una variabile condition).

La CONDITION rappresenta una coda nella quale i processi si sospendono. Le operazioni consentite sono:

- **wait(cond):** sospende il processo introducendolo nella coda individuata dalla variabile cond e il monitor viene liberato;
- **signal(cond):** riattiva un processo in attesa nella coda individuata dalla variabile cond.

Se un processo Q invia una Signal(Cond) a P concettualmente sono possibili due strategie:

- SIGNAL_AND_WAIT: il processo P riprende immediatamente l'esecuzione (è il primo ad operare nel monitor) ed il processo Q viene sospeso nella coda dei processi che attendono di usare il monitor (entry queue);
- SIGNAL_AND_CONTINE: il processo Q prosegue la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver risvegliato il processo (P deve comunque ritestare le condizioni prima di rientrare nel monitor).

E' anche possibile risvegliare tutti i processi sospesi sulla variabile condizione utilizzando la SIGNAL_ALL (variante della signal_and_continue).

JAVA THREAD PARTE II 10.0.0

Il problema della coda unica presente nella sincronizzazione mediante wait e notify, è stato superato con l'introduzione delle **variabili condizione** definite in java.util.concurrent.locks .

```
Public interface Condition {
      void await() throws InterruptedException;
      void signal();
      void signalAll();
}
```

Dove la semantica di signal() è **signal_and_continue.**
Inoltre dalla versione 5.0 , oltre ai blocchi synchronized, java permette di poter utilizzare esplicitamente il concetto di lock definito in java.util.concurrent.locks.

```
Public interface Lock{
      void lock();
      void unlock();
      Condition newCondition();
}
```

Ad ogni variabile condizione deve essere assegnato un lock che:
- Viene liberato quando il thread viene sospeso tramite **await();**
- Viene rioccupato quando il thread viene risvegliato tramite **signal()** o **signalAll().**

La creazione della variabile condition, invece, si effettua tramite il metodo **newCondition()** del lock ad essa associato.

ES.

```
      Lock lock = new ReentrantLock();
      Condition c = lock.newCondition();
```

Dove **ReentrantLock** è una delle possibili implementationi dell'interfaccia Lock.

Tutti questi strumenti concorrono alla definizioni di classi che rappresentano **monitor**, e cioè:

Un lock per la mutua esclusione, da associare a tutte le variabili condizioni;

- Variabili condizioni;
- Variabili interne: stato delle risorse gestite;
- Metodi public ("entry"), privati e costruttore.

11.0 BASH, FILE COMANDI UNIX

Shell: Programma che permette di far interigarire l'utente con il sistema operativo tramite comandi.
In realtà la shell è un interprete di comandi evoluto con un potente linguaggio di scripting e che interpreta comandi ed esegue da standard input o da file comandi.
Esempi di shell: Bourne shell, C shell, Korn shell.
L'implelementazione della **bourne shell** in Linux è il **bash.**

11.0.1 COMANDI SHELL LINUX: FILTRI

Alcuni esempi di comandi shell linux:

`grep <testo> [<file>...]`: ricerca di testo. Input: lista di file. Output: video.
`tee <file>`: scrive l'input sia su file sia su canale di output `sort [<file>...]`: ordina alfabeticamente le linee. Input (lista di) file. Output: video.
`rev <file>`: inverte l'ordine delle linee di file. Output: video. `Cut [-options] <file>`: seleziona colonne da file. Output: video.

11.0.2 PIPING

L'output di un commando può essere diretto a diventare l'input di un altro commando (piping).
Esempi di piping:

`who | wc -l` : conta gli utenti collegati.
`ls -l | grep ^d | rev | cut -d' ' -f1 | rev`
mostra i nomi delle sottodirectory della directory corrente.

- `ls -l` lista i file del direttorio corrente
- `grep` filtra le righe che cominciano con la lettera d (pattern ^d, vedere il man) ovvero le directory (il primo carattere rappresenta il tipo di file)
- `rev` rovescia l'output di grep
- `cut` taglia la prima colonna dell'output passato da rev, considerando lo spazio come delimitatore (vedi man)

✓ quindi, poiché `rev` ha rovesciato righe prodotte da ls -l, estrae il nome dei direttori 'al contrario'

- `rev` raddrizza i nomi dei direttori

11.0.3 METACARATTERI

La shell riconosce caratteri speciali, detti anche wild card, un esempio sono:

- ***** una qualunque stringa di zero o più caratteri in un nome di file
- **?** un qualunque carattere in un nome di file
- **[zfc]** un qualunque carattere in un nome di file, compreso tra quelli nell'insieme. Anche range di valori: [a-d]
- **#** commento fino alla fine della linea. \ escape: segnala di non interpretare il carattere successivo come speciale.

Per esempio:

`ls [q-s]*` lista i file con nomi che iniziano con un carattere compreso tra q e s. `ls [ap, 1-7]*[c,f,d]?` Elenca i file i cui nomi hanno come iniziale un carattere compreso tra a e p oppure tra 1 e 7 e il cui penultimo carattere sia c, f oppure d. `ls *\**` elenca i file che contnegono in qualunque posizione il carattere *

11.0.4 VARIABILI

In ogni shell è possibile deifnire un insieme di variabili, trattate come stringhe con nome e valore. I riferimenti delle variabili si fanno con il carattere speciale **$ ($nomevariabile).**
Si possono fare anche assegnamenti:
`nomevariabile=$nomevaraibile.`
Esempio:

```
x=2
```

`echo $X` (visualizza 2)

`echo $PATH` (mostra il contneuto della variabile PATH)

`PATH=/usr/local/bin:$PATH` (aggiunge la directory /usr/local/bin alla directory di default del path).

Per vedere tutte le variabili di ambiente e i valori associati si può utilizzare il comando **set**:

```
$ set
BASH=/usr/bin/bash
HOME=/space/home/wwwlia/www
PATH=/usr/local/bin:/usr/bin:/bin
PPID=7497
PWD=/home/Staff/AnnaC
SHELL=/usr/bin/bash
TERM=xterm
UID=1015
USER=anna
```

11.0.5 ESPRESSIONI

Le variabili shell sono stringhe, quindi è possibile forzare l'interpretazione numera di scrighe che contnegono la codifica di valori numerici.

Comando **`expr`**:

```
expr 1 + 3
```

Esempio:

```
var = 5
echo risultato: $var+1
$var+1
```

11.0.6 ESPANSIONE

Prima dell'esecuzione la shell prepara i comandi come filtri, successivamente, se la shell trova dei caratteri speciali produce delle sostituzioni (passo di espansione).

- Comandi contenuti tra **black quote** ` ` sono eseguiti e sostituiti dal risultato prodotto.
- Nomi delle variabili **$nome** sono espansi nei valori corrispondenti.
- Metacaratteri *** ? []** sono espansi nei nomi di file secondo un meccanismo di pattern matching.
- \ carattere successivo è considerato come un normale carattere
- **‘ ‘** (apici) proteggono da qualsiasi tipo di espansione **“ “** (doppi apici) proteggono dalle espansioni con l'eccezione di $ \ e blackquote.

Esempi:

`rm '*$var'*` rimuove i file che cominciano con *$var

`rm "*$var"*` rimuove i file che cominciano con *<contenuto della variabile var>

11.1.0 SCRIPTING

La shell è un processore comandi in grado di interpretare file sorgenti in formato testo e contenenti comandi. Per rendere eseguibile il file comandi è necessario regolare i permessi usando la chmod.

La prima riga di un file comandi deve specificare quale shell si vuole usare, usando la sintassi

```
#!/bin/bash
```

Dove **#!** È visto dal sistema operativo come una direttiva di interprete, da ciò il SO capisce che questo script sarà in bash.

11.1.1 PASSAGGIO DEI PARAMETRI

```
./nomefilecomandi arg1 arg2 ... argN
```

Gli argomenti sono variabili posizionali nella linea di invocazione contenute enll'ambiente shell.

- $0 rappresenta il comando stesso
- $1 rappresenta il primo argomento

E' possibile far scorrere tutti gli argomenti verso sinistra usando `shift, $0` non va perso, solo gli altri sono spostati ($1 perso)

	$0	**$1**	**$2**
Prima di **shitft**	nomefilecomandi	-w	/usr/bin
Dopo **shitft**	nomefilecomandi	/usr/bin	
...			

E' possibile riassegnare gli argomenti $1 ... $n usando set.

- `set exp1 exp2 exp3 ...`
- Gli arogmenti sono assegnati secondo la posizione
- $0 non può essere riassegnato

Oltre agli argomenti di invocazione del comando:

- **$*** insieme di tutte le variabili posizionali che corrispondono ad argomenti del comando: $1, $2, ecc.
- **$#** numero degli argomenti passati ($0 escluso)
- **$?** Valore (int) restituito dall'ultimo comando eseguito. **$$** id numero del processo in esecuzione (pid)

Forme di I/O

- `read var1 var2 var3 #lettura da stdin`
  - le stringe in ingesso vengono attribuite alle variabili secondo la corrispondenza posizionale
- `echo var1 contiene $var1 e var2 contiene $var2 #output su stdout`

Strutture di controllo

Ogni comando in uscita restituisce un valore di stato che indica il suo completamento o fallimento. Tale valore di uscita è posto nella variabile?

- $? Può essere riutilizzato in espressioni o per controllo di flusso successivo.

Test

Il comando usato per la valutazione di una espressione è:

```
test -<opzioni> <nomefile>
```

Se il valore è zero => true, altrimenti false.

Test

- `-f <nomefile>` esistenza di file
- `-d <nomefile>` esistenza di direttori
- `-r <nomefile>` diritto di lettura sul file (`-w` e `-x`)
- `test <stringa1> = <stringa2>` uguaglianza di stringhe
- `test <stringa1> != <stringa2>` disuguaglianza stringhe

NB gli spazi intorno a **=** e a **!=** sono **NECESSARI**!

Test con valori numerici

- `test <val1> -gt <val2>` `#val1 > val2`
- `test <val1> -lt <val2>` `#val1 < val2`
- `test <val1> -le <val2>` `#val1 <= val2`
- `test <val1> -ge <val2>` `#val1 >= val2`

11.2.0 CICLI

Strutture di controllo: if

```
if <condizioni>; then
                        <comandi>
[elif <condizioni>; then
                        <comandi> ]
[else
                        <comandi> ]
fi
```

```
if <condizioni>  then
                        <comandi>
[elif <condizioni>  then
                        <comandi> ]
[else
                        <comandi> ]
fi
```

Le parole chiave (do, then, fi, ..) devono essere o a capo o dopo il separatore ; (punto e virgola).

Strutture di controllo: case/switch

```
case <var> in
<pattern-1>)
            <comandi>;;
...
         <pattern-n>)
            <comandi>;;
esac
```

Importante: nello switch si possono usare metacaratteri per fare pattern-matching.

Cicli enumerativi: for

```
for <var> [in <list>] #list=lista di stringhe
do
   <comandi>
done
```

Scansione della lista <list> e ripetizione del ciclo per ogni stringa presente nelal lista.
Scrivnedo for i si itera con valori di i in $*. Quindi si itera su parametri in ingresso $1 $2 $3

Esempio:
`for i in * #esegue per tutti i file nella directory corrente.`

Ciclo non enumerativo: do while

```
while <condizione o lista comandi> do
    <comandi> done


_____________________________________________

until <lista-comandi> do
     <comandi> done
```

Usite anomale

- vedi C: `continue, break, return`
- `exit[status]`: system call di UNIX ma anche comando di shell.